UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

CREACIÓN, DISEÑO Y ANÁLISIS DE UN NUEVO SISTEMA DE CODIFICACIÓN DE IMAGEN Y VÍDEO BASADO EN SALTOS SEMÁNTICOS EN EL DOMINIO DEL ESPACIO

MARINA GONZÁLEZ CASQUETE

2014

CREACIÓN, DISEÑO Y ANÁLISIS DE UN NUEVO SISTEMA DE CODIFICACIÓN DE IMAGEN Y VÍDEO BASADO EN SALTOS SEMÁNTICOS EN EL DOMINIO DEL ESPACIO

AUTORA: Marina González Casquete

TUTOR ALCATEL-LUCENT: José Javier García Aranda

PONENTE: Francisco González Vidal

MIEMBROS DEL TRIBUNAL CALIFICADOR:

Presidente: D.Julio José Berrocal

Vocal: D. Gabriel Huecas Fernández - Toribio

Secretario: D. Francisco González Vidal

Suplente: D. Enrique Vázquez Gallo

FECHA DE LECTURA:

CALIFICACIÓN:

RESUMEN

En este proyecto se exponen, por un lado, los fundamentos de un nuevo sistema de codificación de imagen. Este sistema, llamado Logarithmical Hopping Encoding (LHE) codifica cada píxel de la imagen utilizando saltos logarítmicos en el dominio del espacio, es decir, trabajando con los valores de luminancia y crominancia de los píxeles, sin necesidad de pasar al dominio de la frecuencia. Además, se realiza el análisis de dicho sistema y se ofrecen resultados comparativos con formatos de compresión actuales, tales como JPEG.

Por otro lado, se presentan las primeras ideas para el desarrollo de un sistema que permita comprimir vídeo utilizando la tecnología LHE. Así mismo, se muestran los primeros resultados obtenidos y las conclusiones derivadas de los mismos.

PALABRAS CLAVE

Compresión de imagen, procesamiento digital de imagen, luminancia, crominancia, compresión de vídeo, códec de vídeo, vectores de movimiento, LHE, JPEG, JPEG2000, PSNR.

AGRADECIMIENTOS

A mis padres, por haberme dado la oportunidad de estudiar y por su apoyo incondicional durante todos estos años.

A Jorge, por saber animarme en los momentos más duros y por tener un abrazo preparado para mí cada vez que volvía a casa.

A mis amigos, los que siempre han estado ahí, por sus ánimos, ayuda y esos pequeños momentos llenos de risas.

A mis tutores, José Javier y Francisco, porque, sin ellos, este proyecto no hubiese sido posible.

Para mis padres

ÍNDICE

GLOSARIO

LHE: Logarithmical Hopping Encoding

VLC: Variable Length Coding

EBCOT: Embedded Block Coding with Optimal Truncation

RD: Rate-Distorsion

MSE: Mean Squared Error

PSNR: Peak Signal-To-Noise Ratio

RLC: Run Lengtn Coding

BMP: Bitmap

GIF: Graphics Interchange Format

PNG: Portable Network Graphics

DPCM: Differential Pulse Code Modulation

ADPCM: Adaptative Differential Pulse Code Modulation

DCT: Discrete Cosine Transform

JPEG: Joint Photographic Experts Group

DWT: Discrete Wavelet Transform

USC-SIPI: University of Southern California-Signal and Image Processing Institute

1 Introducción

Hiparco de Nicea (c.190 a.C- c.120 a.C) fue un amante de la astronomía, geografía y las matemáticas de su tiempo. Durante los años que vivió, trabajó duramente y realizó numerosos estudios en estos campos. Una de las aportaciones más importantes que realizó fue el primer catálogo de estrellas realizado. Éste contenía la posición en coordenadas eclípticas de 1080 estrellas diferentes. Con el propósito de elaborar dicho catálogo, Hiparco inventó instrumentos tales como el teodolito que permitía indicar posiciones y magnitudes de forma que fuese fácil saber si las estrellas morían o nacían, si modificaban su posición o cambiaba su brillo. Además, realizó una clasificación de estas estrellas según la intensidad de su brillo, de su luminancia.

Figura 1. Hiparco de Nicea

No fue hasta muchos años más tarde cuando Ernst Heinrich Weber (1795-1878) propuso una ley en la que se establecía la relación cuantitativa entre la magnitud de un estímulo físico (la intensidad luminosa) y cómo éste es percibido. Dicha ley fue elaborada hasta su forma actual por Gustav Theodor Fechner (1802-1887) dando lugar a la Ley de Weber Fechner. Esta ley indica que el ojo humano percibe variaciones en la intensidad luminosa únicamente cuando se producen variaciones logarítmicas del brillo. Es decir, Hiparco de Nicea, ya en la Antigüedad y sin saberlo, clasificó las estrellas según la variación logarítmica del brillo que emitían.

En este proyecto, seremos como Hiparco de Nicea pero, en lugar de clasificar estrellas, trataremos de codificar el brillo de los píxeles de una imagen de forma espacial, es decir, sin utilizar transformaciones frecuenciales como la que se realiza, por ejemplo, con la Discrete Cosine Transform (DCT), utilizada por uno de los algoritmos de compresión de imagen más extendidos: JPEG. Nuestro objetivo será, por tanto,

predecir el brillo de un píxel en función de los anteriores y codificar el error cometido en dicha predicción. A la hora de realizar este trabajo, contaremos con una ventaja y es que, al contrario que Hiparco, sabemos que, por tratarse de errores intensidad luminosa, dicho error será únicamente apreciado por el ojo humano si hay variaciones logarítmicas del brillo.

1.1 Objetivos

El objetivo principal de este proyecto es implementar y poner a prueba una nueva tecnología llamada LHE. Dicha tecnología realiza la codificación de imágenes en el dominio del espacio, lo que permite abrir un nuevo camino de desarrollo, lejos de la línea de trabajo los algoritmos de codificación tales como JPEG o JPEG2000 que utilizan transformaciones frecuenciales para realizar la compresión de las imágenes. De esta manera, se pretende explorar un nuevo método que permita realizar nuevas aportaciones en el campo de la codificación de imágenes y vídeo en un mundo en el que los servicios de vídeo interactivo cada vez están más en auge. Es decir, se hace necesario explorar alternativas que mejoren los tiempos de compresión y la calidad obtenida tanto de las imágenes como del vídeo.

De esta manera, se pretende, mediante este nuevo sistema, mejorar el ratio de compresión de forma significativa sobre algoritmos estándar basados en DCT (JPEG), wavelets (JPEG2000) y otros algoritmos predictivos que también trabajan en el dominio del espacio (WebP) a la vez que se consigue una compresión mayor de la imagen. Todo ello, con menor coste computacional y, por tanto, logrando menores tiempos de compresión que permitan el uso de este algoritmo, además de en imagen, en sistemas de vídeo interactivo. Para ello, se tratará de relacionar secuencias de imágenes (*frames*) también en el dominio del espacio, tratando de reducir el tiempo invertido en cálculo de vectores de movimiento que hace que algoritmos actuales como H.264 vean muy limitado su uso en servicios tales como cloud gaming, en los que el tiempo de codificación y decodificación es vital para una buena experiencia de usuario.

Por último, el algoritmo no debe estar sujeto a patentes puesto que limitaría su difusión, como ocurre con el algoritmo JPEG2000, que obtiene mejores resultados en calidad de imagen que JPEG pero no está tan extendido debido a dichas patentes.

1.2 Estructura de esta memoria

Esta memoria relata el desarrollo de un nuevo algoritmo de codificación de imagen y vídeo que pretende mejorar la compresión y la calidad conseguidas en la imagen final con respecto a los sistemas anteriores. Por esta razón, en el Estado del Arte se comienza haciendo un recorrido por la historia en el que se verán las tecnologías y algoritmos más importantes que han ido apareciendo a lo largo de los años. En primer lugar, se detallarán algunos de los algoritmos de compresión genéricos existentes. Dichos algoritmos permiten reducir el tamaño final de un fichero utilizando diversas técnicas que se explican en los apartados correspondientes. A continuación, se introducirán los conceptos relacionados con el proceso de codificación de la imagen además de mostrar diferentes técnicas y formatos existentes. Para finalizar el Estado del Arte, se introducirán la historia y los conceptos relacionados con la codificación de vídeo entre los que se encuentran la estimación de vectores de movimiento.

Una vez mostrados los sistemas, técnicas y formatos más importantes hasta la fecha, se procederá a detallar los fundamentos de un nuevo algoritmo de codificación de imagen denominado Logarithmical Hopping Encoding (LHE). Como se verá en estos apartados, este algoritmo pretende romper la tendencia de los algoritmos desarrollados hasta la fecha trabajando en el dominio espacial en lugar de en el frecuencial, tal y como hacen algoritmos tan extendidos como JPEG. A continuación, se mostrará la arquitectura del sistema software desarrollado así como los resultados experimentales obtenidos con el códec implementado.

Por otro lado, el citado algoritmo también permite codificar vídeo. Por esta razón, se introducirán los fundamentos que asientan la base para el desarrollo de un nuevo códec de vídeo y se mostrarán los primeros resultados obtenidos en los experimentos.

Para finalizar, se detallarán las conclusiones que derivan de todo el desarrollo realizado así como el trabajo futuro a realizar para que esta nueva tecnología pueda seguir creciendo.

2 Estado del arte

2.1 Algoritmos de compresión genéricos

2.1.1 *Run Lenght Coding* (RLC)

RLC es un algoritmo de compresión de datos muy sencillo y que es utilizado por muchos formatos de archivo de imagen tales como TIFF, BMP o PCX. Este algoritmo puede emplearse para comprimir cualquier tipo de información pero el ratio de compresión que consigue está limitado a la forma en la que esta se distribuye. Esto se debe a que RLC trata de agrupar secuencias de datos con el mismo valor que se repiten y las almacena como un único valor (el que se repite) y el recuento de dicho valor. De esta manera, cuantos mayores sean estas secuencias de datos, mayor será la compresión conseguida.

Así, la secuencia que se quiere codificar se conoce como *run* y, normalmente, se emplean dos bytes para representarla: el primer byte indica el número de caracteres que existen en el *run* y se denomina *run count* mientras que el segundo byte indica el valor del carácter que se está repitiendo y se denomina *run value*. A continuación, vemos un ejemplo:

1. Tenemos la siguiente secuencia de información:

   BBBBBBBBBBBBBBBNNNNNNNNNNNNNNNRRRRRRRRRRRRRRR

2. RLC mira las secuencias que se repiten y las agrupa. De esta manera, se codificaría:

   15B 15N 15R

Las ventajas de esta forma de codificación son evidentes: se trata de un algoritmo simple, fácil de implementar y su ejecución es muy rápida. Sin embargo, en muy raras ocasiones conseguirá mejores ratios de compresión que otros algoritmos de compresión más avanzados.

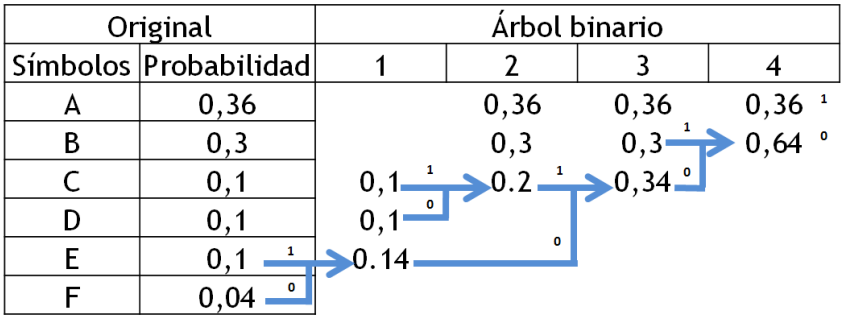
2.1.2 Huffman

Huffman es uno de los algoritmos más conocidos que permite eliminar redundancia de datos. Fue creado por Huffman en el año 1952 cuando su profesor, Robert. M. Fano, anunció que, para superar la asignatura de Teoría de la Información, sus alumnos tendrían o bien que superar un examen o bien presentar un trabajo en el que se explicase un código binario más eficiente. Huffman no consiguió demostrar qué código binario era más eficiente por lo que comenzó a estudiar para el examen. Sin embargo, un día,

mientras estudiaba, se le ocurrió la idea de agrupar los símbolos según la frecuencia de repetición y, así, crear un árbol binario que asignase códigos a cada rama.

De esta manera, el algoritmo de Huffman ordena los símbolos objeto de la codificación según la probabilidad de ocurrencia y va combinando aquellos que tienen la probabilidad más pequeña. Es decir, se crea una rama cuya probabilidad es la suma de las probabilidades de los símbolos. Así, combinando ramas y símbolos, se va creando un árbol en el que cada rama es la suma de las dos probabilidades más bajas en el árbol en ese momento. Este proceso se realiza hasta que únicamente quedan dos ramas. Una vez finalizado el árbol, puesto que las ramas se han sumado dos a dos, a cada ramificación se le asignará un 1 o un 0. De esta manera, el código será más pequeño en las últimas ramas del árbol y será mayor en las más pequeñas, ya que se van concatenando los 1 y 0 según las ramas que se hayan creado para llegar a ese símbolo.

En la Figura 2 vemos un ejemplo de cómo se construye el árbol binario. La asignación de código a los símbolos se realizaría desde 4 hasta 1, concatenando los 1 y los 0 de cada rama hasta alcanzar el símbolo deseado, antes de combinarlo con otro. En la Figura 2 observamos los códigos resultantes.

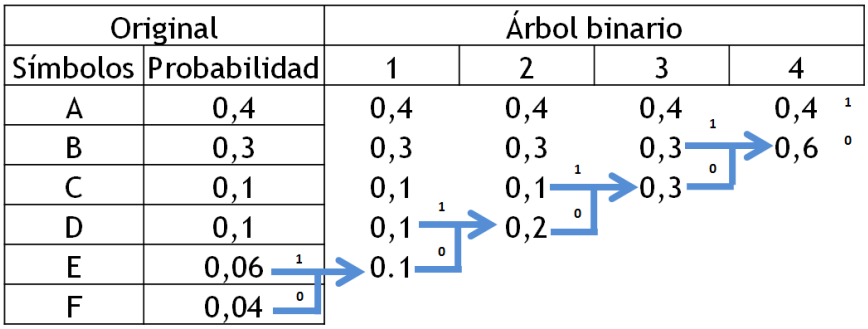


Símbolos	A	B	C	D	E	F
Código	1	01	0011	0010	0001	0000

Figura 2. Árbol binario y códigos Huffman

En el ejemplo anterior observamos que tanto los códigos asignados a los símbolos C y D, como los asignados a E y F tienen la misma longitud. En el caso de E y F se debe a que, al ser los dos últimos símbolos de la tabla Huffman, siempre tendrán la misma longitud (estos dos últimos símbolos serán los de menor probabilidad). Sin embargo, en el caso de C y D se debe a que la distribución de probabilidades es

tal que el hecho de asignar códigos de la misma longitud logra mejores ratios de compresión. Sin embargo, existe un caso de Huffman en el que la distribución estadística es extrema por lo que todos los códigos tienen una longitud diferente (con excepción de los dos símbolos menos probables). En la Figura 3 se observa un ejemplo de este caso. Como ya veremos al describir LHE, los símbolos resultantes de la codificación suelen tener una distribución estadística extrema.



Símbolos	A	B	C	D	E	F
Código	1	01	001	0001	00001	00000

Figura 3. Árbol binario y códigos Huffman con compresión estadística extrema

2.1.2.1 Códigos Huffman *n*-arios

Es posible crear códigos Huffman n-arios. Para ello, es necesario introducir algunas modificaciones en el algoritmo de Huffman básico.

- La primera modificación consiste en que, en lugar de sumar dos a dos los símbolos, se sumarán *n* a *n*, generando *n* ramas cada vez que se combinan símbolos.

- La segunda modificación consiste en que, si tenemos m símbolos en el alfabeto a codificar, (m-1) tendrá que ser múltiplo de (*n*-1) siendo *n* el número de símbolos empleados en el código Huffman.

- Si la segunda modificación no se cumple, entonces se tendrán que añadir símbolos con frecuencia 0 hasta que ésta se cumpla para poder crear el árbol.

2.1.2.2 Código Huffman adaptativo

Se trata de una variante Huffman directamente relacionada con la familia de algoritmos Lempel-Ziv. De esta manera, las ramas del árbol Huffman se modifican según van apareciendo los símbolos, de una forma similar a cómo se crean los diccionarios en los algoritmos Lempel-Ziv. La diferencia fundamental con el algoritmo de Huffman básico es que no es necesario realizar un estudio previo de la distribución estadística de los símbolos sino que el árbol se modifica a medida que se codifica la cadena de símbolos. Esto, además, garantiza mejores ratios de compresión debido a la alta correlación que existe entre los símbolos en un archivo.

2.1.3 Algoritmos Lempel-Ziv

Los algoritmos Lempel-Ziv que aquí se tratan permiten realizar una codificación de los símbolos sin necesidad de haber realizado un estudio estadístico de los mismos. Esto lo consiguen mediante la creación de un diccionario en el que se introducen nuevas entradas que contienen las cadenas de símbolos que van apareciendo en el archivo a medida que se codifica la cadena de símbolos. El buen funcionamiento de los algoritmos que se explican a continuación se debe al hecho de que la información de un archivo está correlada entre sí por lo que es más probable que aparezca un símbolo acompañado de otros (la misma cadena en cada aparición) repetidamente a que aparezca aislado.

2.1.3.1 LZ78

Este algoritmo fue ideado por Abraham Lempel y Jacob Ziv en el año 1978. La compresión se realiza gracias a la creación de un diccionario que se va rellenando a medida que se codifica. Para realizar dicha acción, se sigue el siguiente procedimiento:

1. En un primer momento, el diccionario está vacío y no se ha tomado ningún carácter del flujo de datos a codificar

2. Se toma, del flujo de datos a codificar, el símbolo siguiente al último que ha sido codificado.

3. Se busca en las entradas del diccionario si ese símbolo ya había aparecido con anterioridad.

    a. Si no había aparecido se genera una nueva entrada en el diccionario que contenga dicho símbolo.

    b. Si había aparecido, se toma el símbolo siguiente del flujo de datos de entrada. A continuación, se comprueba si este conjunto de símbolos existe en el diccionario. Si existe, se continúa tomando símbolos del flujo de datos hasta que no haya ninguna

entrada coincidente. En ese momento, se genera una nueva entrada en el diccionario con el conjunto de *n* símbolos que se ha obtenido.

4. Por último, se codifica el conjunto de *n* símbolos como un par {(índice), (símbolo *n* del conjunto)} donde índice es un número de entrada anterior en el diccionario que coincide con los *(n*-1) primeros símbolos del conjunto.

De esta manera, si tenemos la cadena ABCBCAABCB se generará el diccionario que aparece en la Tabla 1. Por tanto, se codificará dicha cadena como: {0, A} {0, B} {0, C} {2, C} {1, A} {4, B}.

Índice	0	1	2	3	4	5	6
Símbolos		A	B	C	BC	AA	BCB

Tabla 1. Diccionario

2.1.3.2 Lempel-Ziv-Welch

Lempel-Ziv-Welch (LZW) es un algoritmo propuesto por Terry Welch en el año 1984 como una versión mejorada del algoritmo LZ78. Las mejoras son las siguientes:

1. Únicamente los símbolos del flujo de datos pueden ser entradas del diccionario. Es decir, no puede existir una entrada vacía. De esta manera, el diccionario debe ser inicializado antes de comenzar el algoritmo. Para ello, se anotan todos los símbolos individuales que existen en el alfabeto del flujo de datos a codificar.

2. Puesto que se han añadido todos los símbolos individuales al diccionario, en el proceso de codificación se comprobarán conjuntos de símbolos de longitud 2 o mayor ya que cualquier símbolo de longitud 1 ya se encuentra en el diccionario.

3. El símbolo que se comienza a comprobar es el último que ya se codificó en el anterior conjunto. Esto es necesario para que sea posible regenerar el diccionario en el decoder.

4. Los conjuntos de símbolos no se codifican como pares, sino con el índice de la entrada del diccionario que contiene dicho conjunto.

De esta manera, para codificar la cadena ABCBCAABC se tendría un diccionario inicializado con los valores:

1. A
2. B
3. C

De esta manera, el proceso de codificación iría generando el código de salida y el diccionario que se muestra en la Tabla 2. Lógicamente, la primera entrada del diccionario que se rellena es la 4 puesto que las anteriores corresponden a la inicialización del diccionario.

Datos	**Codificación**	**Diccionario**	**Símbolos**
AB	1	4	AB
BC	2	5	BC
CB	3	6	CB
BCA	5	7	BCA
AA	1	8	AA
ABC	4	9	ABC
C	3	-	-

Tabla 2. Diccionario y símbolos

Por otro lado, en el decodificador, el diccionario se ha inicializado de la misma manera que en el codificador. A medida que se recibe el código, se decodifica y se regenera el diccionario tal y como indica la Tabla 3.

Código recibido	Código decodificado	Diccionario	Símbolos
1	A	-	-
2	B	4	AB
3	C	5	BC
5	BC	6	CB
1	A	7	BCA
4	AB	8	AA
3	C	9	ABC

Tabla 3. Diccionario y símbolos en el decodificador

Este procedimiento logra mejores resultados que el utilizado por el algoritmo LZ78 puesto que el diccionario de caracteres se crea de una manera más eficiente. De hecho, LZ78 necesita codificar los símbolos utilizando un par de valores (índice de la entrada del diccionario y símbolo a añadir al conjunto tomado de dicha entrada) mientras que el desarrollo de LZW permite codificar indicando únicamente la entrada del diccionario donde se encuentra el símbolo correspondiente.

2.2 Codificación de imágenes

2.2.1 Conceptos

2.2.1.1 Complejidad computacional

Un algoritmo es un procedimiento computacional bien definido que considera un conjunto de valores de entrada y, a través de una secuencia de instrucciones organizadas produce un conjunto de valores de salida. El objetivo de los algoritmos es dar solución a un problema específico.

Así, la complejidad computacional estudia la eficiencia de los algoritmos. Para ello, se evalúa su efectividad de acuerdo al tiempo de ejecución y a la memoria requerida en la máquina que lo ejecuta. De esta manera, la viabilidad de implementación de dicho algoritmo es evaluada en términos de tiempo y de costo. Para evaluar el tiempo de ejecución, se coge el mayor tiempo que tardaría el algoritmo en obtener los valores de salida.

2.2.1.2 Análisis asintótico de los algoritmos

El análisis asintótico permite estudiar la eficiencia de un algoritmo teniendo en cuenta el tiempo de ejecución cuando el tamaño de los datos de entrada (n) es suficientemente grande, de tal forma que las constantes y los términos de menor orden no afectan. De esta manera, para expresar la tasa de crecimiento de una función se tiene en cuenta:

- El término dominante con respecto a n
- Se ignoran las constantes.

Para expresar el tiempo de ejecución asintótico de un algoritmo se definen diferentes notaciones en términos de funciones cuyo dominio es el conjunto de números naturales. De esta manera, se puede describir la función del peor caso del tiempo de ejecución T(n). Existen diferentes tipos de notación:

- Notación Θ: dada una función g(n), se denota por Θ(g(n)) al conjunto de funciones tales que

$$\Theta\big(g(n)\big) = \{f(n): \exists\, c_1, c_2\,, n_0 > 0 |\ 0 \leq c_1 g(n) \leq f(n) \leq\ c_2 g(n)\ \forall\ n > n_0$$

  En la Figura 4 vemos un ejemplo de este comportamiento asintótico.

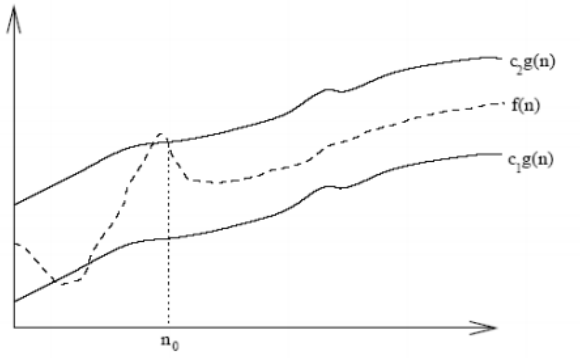


Figura 4. Comportamiento asintótico

- Notación O: representa una cota asintótica superior. Ses una función g(n), se denota por O(g(n)) el conjunto de funciones:

$$O\big(g(n)\big) = \{f(n): \exists\, c, n_0 > 0 \mid 0 \leq f(n) \leq cg(n)\ \forall\ n > n_0$$

La notación O se utiliza para acotar el peor caso de tiempo de ejecución de un algoritmo. En la Figura 5 se observa un ejemplo de este comportamiento asintótico.

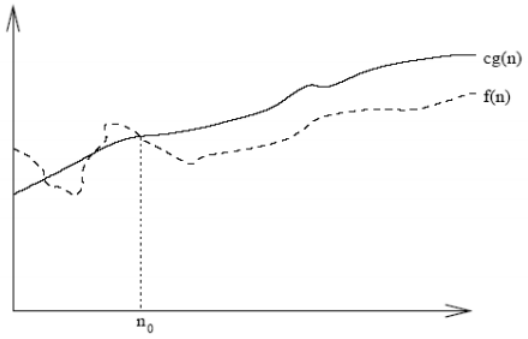


Figura 5. Comportamiento asintótico

2.2.1.3 *Lossy* vs *Lossless*

Se dice que un algoritmo de compresión de imágenes es *lossless* cuando su objetivo es reducir el número de bits que se emplean en la codificación de la imagen sin que se pierda información de la original. Es decir, el decodificador debe ser capaz de reconstruir la imagen original perfectamente, sin ninguna

pérdida de información. Los algoritmos *lossless* son útiles cuando una aplicación requiere que la imagen original y decodificada sean exactamente iguales como, por ejemplo, las imágenes médicas o las imágenes en astronomía.

Por otro lado, existe otra serie de algoritmos que admiten pérdida de información. Estos algoritmos se denominan *lossy* y se basan en la tolerancia que tiene el ojo humano a ciertos errores en una imagen de forma que la percepción de la misma no varía con respecto a la original. Así, la imagen generada en el decodificador tiene errores con respecto a la original. Sin embargo, se puede lograr una aproximación suficientemente buena como para que el ojo humano no detecte los errores que se han cometido.

Los algoritmos con pérdidas consiguen mejores ratios de compresión que los algoritmos sin pérdidas ya que los primeros permiten eliminar información de la imagen original por lo que no tiene que ser almacenada y el tamaño final es menor. Sin embargo, la imagen resultante de la compresión verá reducida su calidad con respecto a la original. Por tanto, es importante que los algoritmos *lossy* mantengan un compromiso entre calidad y compresión de forma que la imagen final sea aceptable para la aplicación deseada. Dicho compromiso se representa mediante la gráfica RD (Rate-Distorsión) que muestra el PSNR (Peak Signal-to-Noise Ratio) en función de los bits por píxel (bpp) de la imagen. El PSNR es una medida que indica lo que se parece una imagen que ha sido sometida a un algoritmo *lossy* con respecto a la original. Dicho de otra manera, muestra cómo de buena es la aproximación que se ha conseguido al aplicar el algoritmo de compresión. El PSNR se calcula por medio del MSE (Mean Squared Error) para una imagen sin ruido I y su aproximación K, ambas de mxn píxeles, tal y como muestra la siguiente ecuación, donde MAX_I es el máximo valor que puede tomar un píxel de la imagen.

$$\text{MSE} = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

$$PSNR = 10x\log_{10}\frac{MAX_I^2}{MSE}$$

Además del PSNR, existen otras medidas de calidad tales como SSIM Y MSSIM que pretenden mejorar la medida dada por el PSNR. Esto se debe a que el PSNR, tal y como se puede observar en la Ecuación que lo define, evalúa los errores cometidos en cada píxel pero no es capaz de diferenciar dónde está situado ese píxel. Es decir, pueden existir errores en la imagen que no son realmente relevantes para el observador ya que se producen en zonas de la imagen con mucho ruido. De esta manera, el PSNR no es capaz de distinguir la calidad subjetiva percibida por el observador y evaluará, de la misma manera, este

error que otro que se haya cometido en un píxel que se encuentra en un área relevante para el observador. De esta manera, SSIM y MSSIM permiten evaluar la calidad subjetiva de una imagen y ofrecen, como resultado, un número entre 0 y 1. Por ejemplo, Google ha ofrecido los resultados de su nuevo formato de imagen WebP utilizando la medida SSIM ya que afirman, al igual que otros autores, que el PSNR no es adecuado para evaluar la calidad de una imagen.

2.2.2 Algoritmos y formatos de compresión de imagen

2.2.2.1 *Run Lenght Coding* (RLC) en imágenes

El algoritmo RLC, explicado en el apartado 2.1, se puede utilizar también en la compresión de imágenes. Así, RLC trata la imagen como un stream de 1 dimensión que comienza en la esquina superior izquierda de la imagen y la recorre. De esta manera, busca secuencias de luminancias que se repiten y las codifica tal y como se indicó en 2.1. Existen diversas alternativas para este proceso de recorrer la imagen ya que, dependiendo de cómo se distribuya la información, se conseguirán mejores ratios de compresión eligiendo la alternativa adecuada. De este modo, una imagen que consista en un degradado vertical logrará un mayor ratio de compresión usando el método a de la Figura 6 puesto que las luminancias serán más parecidas en la dirección horizontal. Por el contrario, una imagen con un degradado horizontal tendrá unas luminancias similares en la dirección vertical y se logrará un mayor ratio de compresión con el método b de la Figura 6.

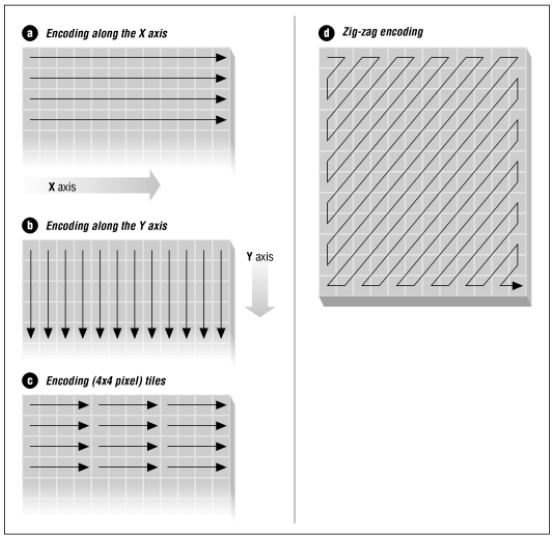


Figura 6. Métodos para recorrer la imagen

2.2.2.1.1 Lossy RLC

RLC es, normalmente, un algoritmo lossless. Sin embargo, lossy RLC es una variante del algoritmo que consiste en transformar la luminancia de ciertos píxeles para que no se rompa una secuencia de luminancias iguales y, de esta manera, se pueda mejorar el ratio de compresión. Si se escogen adecuadamente estos píxeles, la pérdida de información no será apreciable en imágenes en las que las luminancias fluctúan de forma muy sutil y el ratio de compresión mejorará notablemente.

2.2.2.2 *Bitmap* (BMP)

El formato BMP, también conocido como bitmap, es utilizado para almacenar archivos de imagen. Se introdujo por primera vez en el sistema operativo Windows aunque actualmente se reconoce este formato en otros sistemas.

BMP perite almacenar la información de la imagen usando diferente número de bits para definir cada píxel:

- 1-bit por píxel (1bpp): permite almacenar dos colores diferentes (generalmente, blanco y negro).
- 2-bit por píxel (2bpp): permite almacenar 4 colores diferentes
- 4-bit por píxel (4bpp): permite almacenar 16 colores diferentes
- 8-bit por píxel (8bpp): permite almacenar 256 colores diferentes
- 16-bit por píxel (16bpp): permite almacenar 65536 colores diferentes
- 24-bit por píxel (16bpp): permite almacenar 16.777.216 colores diferentes
- 32-bit por píxel (16bpp): permite almacenar 4. 294.967.296 colores diferentes

Como se puede observar, BMP no aplica ningún algoritmo de compresión sino que reduce la cantidad de colores que almacena. Pese a todo, el mínimo número de bits por píxel que dedica es 2bpp y, pese a que se trata de un ratio de compresión considerable, la imagen obtenida será muy pobre puesto que solo acepta dos valores diferentes de color. Sin embargo, los archivos BMP de 24 y 32 bit por píxel logran imágenes de gran calidad, sin errores visibles de codificación. No obstante, los archivos generados son muy pesados. De esta manera, BMP no puede ser usado en aplicaciones que requieran del almacenamiento o descarga de imágenes.

2.2.2.3 Graphics Interchange Format (GIF)

GIF es un formato de imagen bitmap que fue introducido por la compañía CompuServe en el año 1987. Desde entonces, su uso se ha expandido por la Web debido a su soporte y portabilidad.

GIF soporta imágenes de hasta 8 bits por píxel es decir, una imagen puede tener una paleta de hasta 256 colores diferentes que se toman del espacio de color RGB. Además, GIF permite realizar animaciones en las que cada *frame* utiliza 256 colores. El hecho de que el formato GIF acepte solamente un máximo de 256 colores hace que no sea apropiado para fotografías. Sin embargo, es muy útil para gráficos, logos e iconos ya que presentan un número limitado de colores y, además, se reduce su tamaño lo que tiene gran utilidad en páginas Web que contengan logos y gráficos con o sin animación. Además, GIF admite transparencias en sus imágenes.

Para la compresión, el formato GIF utiliza el algoritmo LZW ya que permite reducir el tamaño del fichero sin degradar la calidad visual. Para ello, se recorre la imagen en horizontal desde la esquina izquierda superior de la imagen y se transforma cada píxel, mediante el algoritmo LZW, en un código que posteriormente se traducirá a un código binario.

2.2.2.4 Portable Network Graphics (PNG)

Se trata de un algoritmo de compresión sin pérdidas para imágenes bitmap no sujeto a patentes que se ideó para solventar los problemas de la compresión GIF. El método de compresión que utiliza PNG tiene dos pasos:

1. Filtrado: se trata de una predicción del valor del píxel. Se elige, para cada línea de la imagen, un tipo de filtro que permite realizar una predicción del valor del píxel, utilizando el valor de los píxeles vecinos y resta la predicción realizada al valor del píxel original. Esta diferencia es la que se codifica. De esta manera, se pretende facilitar el proceso de compresión para que ésta sea más eficiente ya que los valores diferencia calculados son más compresibles que el valor original del píxel sobre todo si una línea de la imagen es similar a la de arriba puesto que los valores diferencia serán cercanos a 0. En la Tabla 4 se resumen los diferentes filtros que se aplican a un píxel X definido en la Figura 7. Estos filtros se aplican al valor en bytes del píxel y no al valor de luminancia como tal.

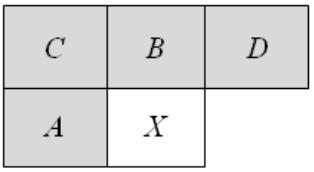


Figura 7. Píxeles

Tipo	Nombre del filtro	Predicción
0	Ninguno	Ninguna (el valor del byte no se altera)
1	*Sub*	Byte A
2	*Up*	Byte B
3	*Average*	Mezcla de los bytes A and B
4	*Paeth*	Se escoge uno de los Byte A, B, or C en función del que tenga el valor más cercano a (A + B – C)

Tabla 8. Comportamiento asintótico

2. Compresión: se utiliza un algoritmo denominado *deflate algorithm* el cual es una mezcla del algorimo LZW sin pérdidas y la codificación Huffman para realizar la codificación de los valores calculados en el paso anterior. Es necesario indicar dónde comienza cada línea ya que este algoritmo no distingue las dos dimensiones de la imagen, solo es capaz de reconocer una cadena de bits.

PNG, comparado con GIF, logra mejores ratios de compresión en las imágenes salvo en imágenes muy pequeñas en las que ocurre lo contrario. Por otro lado, PNG admite formatos con una profundidad de color de 48 bits por píxel en color real lo que proporciona rangos de color más ricos y precisos que GIF el cual admite, únicamente, 256 colores. Por último, es necesario señalar que, mientras el formato GIF soporta animación, PNG no la soporta como tal aunque existen algunas extensiones no oficiales de este algoritmo que sí que lo hacen. Por otro lado, PNG, al igual que GIF admite transparencias pero mientras que GIF únicamente soporta dos opciones (totalmente opaco o totalmente transparente), PNG permite

utilizar mayor profundidad de bits para lograr efectos de semi-transparencia, propios de objetos translúcidos. Por ejemplo, con 8 bits lograríamos 2^8=256 niveles de transparencia.

No se puede realizar una comparación muy detallista entre JPEG y PNG ya que son formatos de imagen creados para fines muy diferentes. De hecho, PNG es un formato *lossless* mientras que JPEG es *lossy*. Sin embargo, es necesario destacar que, mientras que las imágenes de pequeño tamaño (iconos) son destruidas por el ruido JPEG, PNG sí que es capaz de comprimirlas con una calidad aceptable. Además, JPEG no admite ningún tipo de transparencia mientras que, como ya hemos visto, PNG sí.

2.2.2.5 Differential Pulse Code Modulation (DPCM)

DPCM es un método de codificación diferencial que recupera valores de píxeles que ya han sido codificados para realizar una predicción del píxel actual. Esto es posible debido a que cada píxel de una imagen está altamente correlado con los píxeles vecinos. De esta manera para cada píxel, en el emisor, se realiza una predicción basada en las muestras que se han predicho anteriormente. El error e_i que se comete en cada predicción de la señal x_i, se cuantiza y se codifica, dando como resultado $\hat{e}_i$. En el decodificador, se recibe este error cuantizado y se suma a la predicción del píxel realizada con las muestras ya decodificadas. El resultado es una aproximación de la luminancia de la imagen original $\hat{x}_i$ En la Figura 9 se muestra el diagrama de bloques del proceso de codificación y decodificación.

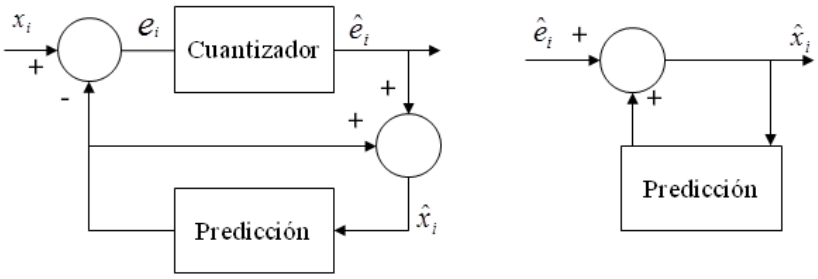


Figura 9. Proceso de codificación y decodificación DPCM

2.2.2.6 Adaptative Differential Pulse Code Modulation (ADPCM)

La idea que subyace en la codificación ADPCM es la misma que en DPCM con la diferencia que los coeficientes del filtro de predicción son modificados con el objetivo de minimizar el error de predicción.

Estos coeficientes se eligen de forma que los diferentes códigos posibles se distribuyen de forma lineal. La Figura 10 muestra un diagrama en el que aparece el codificador y decodificador ADPCM. Esta técnica se detalla en el estándar ITU-T G.726 *speech* códec y también se puede usar para compresión de imágenes [1] aunque no existe ningún formato gráfico estándar que utilice esta técnica.

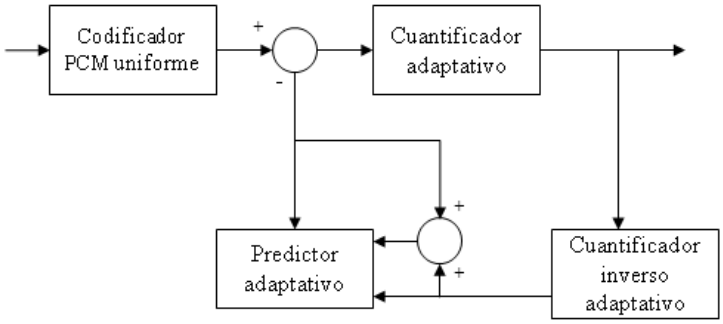


Figura 10. Codificador y decodificador ADPM

2.2.2.7 Algoritmos basados en *Discrete Cosine Transform* (DCT)

La codificación por transformación por DCT se utiliza en el estándar de compresión de imágenes JPEG y, también, en los estándares de compresión de vídeo H.261, H.263, MPEG, MPEG-2 y MPEG-4. Su complejidad computacional viene dada por $0(n) = n \log n$ En la Figura 11, se presenta el esquema de codificación y sus pasos inversos para la decodificación.

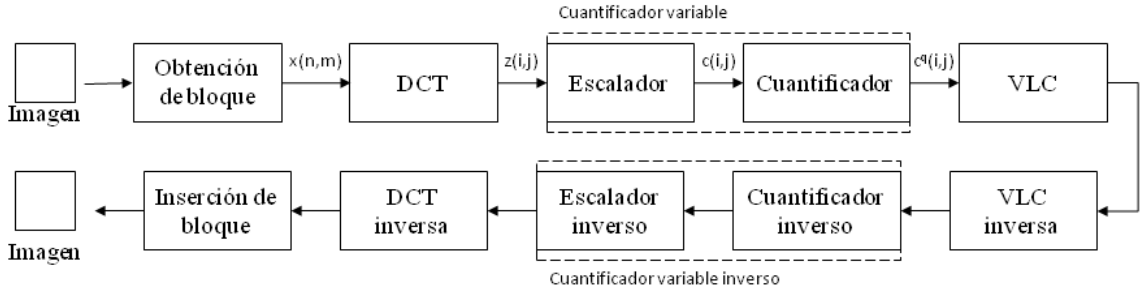


Figura 11. Codificación y decodificación DCT.

Los pasos simplificados que se deben seguir para realizar la codificación de imágenes o de vídeo por medio de la transformada DCT son:

1. La imagen se divide en bloques X no solapados de 8x8 píxeles. Los bloques se transforman en matrices de 64 elementos donde x(n,m) es el valor de luminancia del elemento (n,m) de la matriz.
2. Se aplica la transformada DCT a cada uno de los bloques de la matriz. La DCT descompone cada bloque en un conjunto de funciones base, cada una de ellas con distinta frecuencia espacial. En la DCT de 8x8 existen 64 funciones base diferentes las cuales se pueden observar en la Figura 12. La función que se encuentra en la esquina superior izquierda es la componente DC, en la que se compacta la mayor cantidad de energía. De esta manera, se trata de encontrar los pesos z(i,j) de cada una de las 64 funciones base de forma que la suma escalada por dichos pesos de las diferentes funciones base reconstruya el bloque original X.

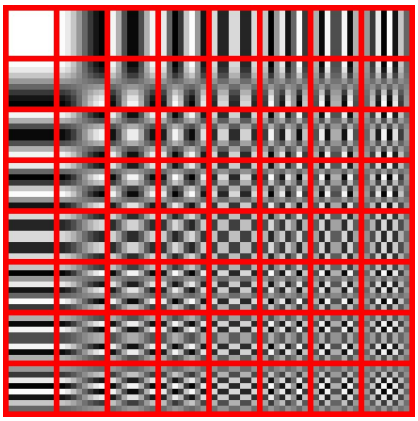


Figura 12. Funciones base

Las expresiones para la transformación DCT directa e inversa para un bloque de 8x8 muestras son:

$$z(i,j) = \frac{1}{4} C_i C_j \sum_{n=0}^{7} \sum_{m=0}^{7} x(n,m) \cos\frac{(2n+1)i\pi}{16} \cos\frac{(2m+1)j\pi}{16} \qquad i,j = 0,1 \dots 7 \qquad (1)$$

$$x(n,m) = \frac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} C_i C_j z(i,j) \cos\frac{(2n+1)i\pi}{16} \cos\frac{(2m+1)j\pi}{16} \qquad n,m = 0,1 \dots 7 \qquad (2)$$

donde:

$$C^k = \begin{cases} \frac{1}{\sqrt{2}} & si\ k = 0 \\ 1 & si\ k \neq 0 \end{cases} \qquad (3)$$

3. En el escalador, cada uno de los coeficientes *z(i,j)* calculados mediante la Ecuación 1 se escalan, es decir, se modifican según unos pesos que dependen del tipo de bloque (por ejemplo, si se trata de componente de crominancia o de luminancia), de la coordenada (i,j) en la que se encuentra el píxel dentro del bloque, la calidad final buscada… Se genera, de esta manera, una matriz de coeficientes c(i,j).
4. En el cuantificador, la matriz de coeficientes c(i,j) se opera con una matriz de cuantificación. Dicha matriz sirve para hacer una división con redondeo al entero más próximo, donde llamaremos q(i,j) al elemento (i,j) de la matriz q. Un ejemplo de matriz se muestra en la Ecuación 4. Esta matriz permite que los coeficientes de más alta frecuencia, es decir, los menos visibles para el ser humano, se cuantifiquen más gruesamente. De esta manera, los elementos del bloque cuantificado C^Q se calculan como: $c^q(i,j) = Entero\ más\ próximo\ \left(\frac{c(i,j)}{q(i,j)}\right)$.

$$q = \begin{Bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{Bmatrix} \quad (4)$$

5. Finalmente, se utiliza un algoritmo tipo VLC (Variable Length Coding) para transformar los coeficientes a bits. Los algoritmos VLC realizan un estudio estadístico de los diferentes símbolos a codificar y asignan los códigos más cortos a los símbolos más probables.

Por último, es necesario mencionar que la transformada DCT no comprime como tal. Lo que hace es descorrelar los datos originales para compactar gran parte de la energía de la señal en un conjunto pequeño de coeficientes. Sin embargo, la energía se preserva por lo que muchos de los coeficientes tendrán muy poca energía. En cuanto a la compresión, se logra en los procesos de cuantificación y codificación. Cuanto mayor sea el grado de cuantificación, es decir, cuanto menos valores se admitan para realizar las aproximaciones, mayor será la compresión obtenida pero la imagen obtenida será de peor calidad. En cuanto a la codificación, se lograrán mayores o menores ratios de compresión dependiendo del método que utilicemos para asignar los bits.

2.2.2.7.1 JPEG

JPEG es un método de compresión con pérdidas utilizado, sobre todo, para la fotografía digital y que utiliza la codificación por transformada DCT para lograr dicha compresión. JPEG es el formato de imagen más común utilizado por las cámaras fotográficas digitales y otros dispositivos de imagen.

JPEG se basa en dos fenómenos visuales del ojo humano. El primero de ellos es que el ojo es que es más sensible a pequeños cambios de brillo en zonas lisas de una imagen (variaciones de baja frecuencia) y menos sensible a los mismos cambios si estos se producen en zonas en las que la imagen presenta variaciones muy grandes (variaciones de alta frecuencia) como, por ejemplo, los bordes de los cuerpos de los objetos. De esta manera, la codificación por transformada DCT permite, tal y como se ha explicado en la sección anterior, descomponer cada bloque de 8x8 de una imagen en 64 funciones base. Cada una de estas funciones estará ponderada con un peso de forma que la suma de dichas funciones ponderadas de cómo resultado el bloque original. Posteriormente, se divide cada uno de estos pesos por los diferentes elementos de una matriz de cuantificación cuyos valores se han elegido para lograr cuantificar más gruesamente aquellas funciones base de frecuencia más elevada, que son a las que el ojo humano es poco sensible. Finalmente, para generar el resultado, se regeneran los bloques pero utilizando los pesos cuantificados es decir, sin las componentes de alta frecuencia.

El segundo fenómeno visual es que el ojo es mucho más sensible al cambio de luminancia que al de crominancia. Por esta razón, JPEG realiza una transformación del espacio de color RGB al espacio YUV y realiza un submuestreo de la crominancia. Las Ecuaciones 4 y 5 muestran cómo se realizan las transformaciones de uno a otro espacio de color. Por otro lado, el submuestreo consiste en tomar menos muestras de cada una de las señales de crominancia UV que de la señal de luminancia Y. De esta manera, se puede reducir la información de crominancia en un factor 2 en la dirección horizontal utilizando el método YUV 4:2:2 de forma que el color tiene la mitad de resolución en horizontal mientras que el brillo sigue intacto. Otro método muy usado es el YUV 4:2:0 que reduce en un factor 2 en ambas direcciones (horizontal y vertical) la información de las señales de crominancia UV mientras que el brillo se conserva.

$$\begin{aligned} Y &= O,257 \times R + 0{,}504 \times G + 0{,}098 \times B + 16 \\ U &= -0{,}148 \times R - 0{,}291 \times G + 0{,}439 \times B + 128 \\ V &= 0{,}439 \times R - 0{,}368 \times G - 0{,}071 \times B + 128 \end{aligned} \tag{4}$$

$$B = 1{,}164 \times (Y - 16) + 2{,}018 \times (U - 128)$$
$$G = 1{,}164 \times (Y - 16) - 0{,}813 \times (V - 128) - 0{,}391 \times (U - 128) \quad (5)$$
$$R = 1{,}164 \times (Y - 16) + 1{,}596 \times (V - 128)$$

JPEG, en general, ofrece buenos resultados en cuanto a compresión y calidad de imagen. Sin embargo, tiene algunas limitaciones:

- Puesto que la transformación DCT se realiza bloque a bloque, aparece un ruido muy significativo en las imágenes JPEG conocido como artefactos. Cuanto mayor sea la compresión, mayor es la presencia de este ruido que se produce porque, al eliminar ciertas componentes de la imagen por bloques, los bloques resultado no son fieles a sus respectivos originales y se aprecian los errores en forma de cuadrados. En la Figura 13 se puede observar un ejemplo de este tipo de ruido.

Figura 13. Ruido JPEG

- JPEG no sirve para codificar imágenes muy pequeñas porque los artefactos los destruyen, solo sirve para imágenes más grandes. En su lugar, los iconos se codifican con PNG o GIF.
- JPEG no soporta transparencias. Si se requiere una imagen con transparencias es necesario usar PNG o GIF.
- La pérdida de calidad en las imágenes JPEG es acumulativa. Es decir, si comprimimos y descomprimimos una imagen esta presentará diferencias con respecto a la original y habrá

disminuido su calidad. Si sobre esta imagen descomprimida volvemos a realizar la compresión-descompresión, la degradación de la imagen será aún mayor.

2.2.2.8 Algoritmos basados en *Discrete Wavelet Transform* (DWT)

La DWT es una herramienta que permite descomponer una señal en sus componentes frecuenciales y espaciales. Es decir, además de descomponer la señal en las diferentes componentes frecuenciales por las que está formada con sus respectivos pesos, las sitúa en sus respectivas posiciones espaciales utilizando un análisis de subbandas. Pese a que se trata de una herramienta muy buena, es muy compleja computacionalmente, tanto en operaciones como en memoria requerida. Por otro lado, la DWT presenta varias ventajas con respecto a la DCT en cuanto a compresión de imágenes:

- Es capaz de compactar la energía en menos coeficientes que la DCT. En el caso de la DCT, la mayor parte de la energía de la señal se compacta en la componente DC y los coeficientes de su alrededor de forma que la mayoría de los coeficientes son 0 y, así, se puede comprimir la señal. En el caso de la DWT, esta concentración de energía en unas pocas componentes se produce más gruesamente por lo que la compresión es aún mayor.
- No es necesario dividir en bloques la imagen para aplicar la DWT puesto que la transformación que realiza sitúa las componentes frecuenciales en el espacio. Esto se traduce en que los artefactos (error de bloques) propios de la aplicación de la DCT por bloques y posterior cuantización de la imagen no aparecen cuando se aplica la DWT por lo que se mejora la calidad subjetiva de la imagen.
- La DWT permite un análisis multiresolución de las imágenes. Esto permitirá incluir fácilmente escalabilidad espacial en un compresor.

Transformada Wavelet 1-D (1D-DWT)

La DWT usa como base funciones de soporte finito, oscilantes que se pueden escalar y desplazar respecto a otra función. Aunque la teoría matemática que respalda la DWT es muy compleja, su implementación es muy sencilla ya se basa en un banco de filtros paso-bajo $\{h_k\}$ y paso-alto $\{g_k\}$ que permiten descomponer una señal unidimensional en dos bandas: una de alta frecuencia y otra de baja frecuencia. A continuación, se hace un *downsampling* de los coeficientes resultantes de forma que sobrevivan la mitad de las muestras obtenidas al aplicar cada filtro. En la Figura 14 aparece el diagrama de este procedimiento.

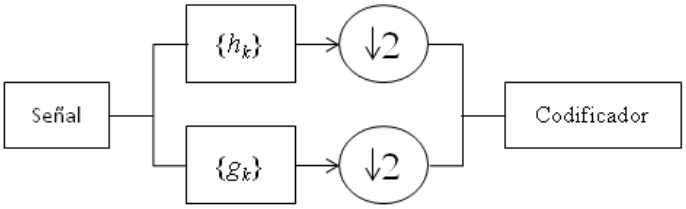


Figura 14. Procedimiento DWT

Uno de los problemas con esta implementación es cómo operar con valores próximos a los bordes puesto que no existen suficientes muestras para aplicar al filtro. Existen varias soluciones a este problema entre las que se encuentra realizar una extensión simétrica de las muestras del borde de forma que se pueda aplicar dicho filtro.

Transformada Wavelet 2-D (2D-DWT)

Para aplicar la DWT a una imagen, es decir, una señal bidimensional, se parte de la transformada 1D-DWT y se aplica, en primer lugar, a cada una de las filas de la imagen y posteriormente a cada una de las columnas o viceversa. De esta manera, aparecen cuatro bandas de frecuencia diferentes:

- LL: aplicación del filtro paso bajo a las filas y a las columnas.
- HL: aplicación del filtro paso alto a las filas y paso bajo a las columnas.
- LH: aplicación del filtro paso bajo a las filas y paso alto a las columnas.
- HH: aplicación del filtro paso alto a las filas y a las columnas.

En la Figura 15 se observa la aplicación de la 2D-DWT a la imagen de Lena.

Figura 15. Bandas DWT de la imagen de Lena

La transformación DWT admite más divisiones de bandas de frecuencia que las anteriormente expuestas. Sin embargo, las cuatro que se han mostrado bastan para entender el procedimiento.

2.2.2.8.1 JPEG2000

JPEG2000 es un compresor de imagen que utiliza la DWT. El procedimiento que sigue es el siguiente:

1. Si la imagen es en color, se realiza una transformación del espacio de color sin comprimir RGB al espacio YUV 4:2:0. Es decir, la luminancia se codifica completa y las componentes de crominancia a la mitad.
2. Se aplica la versión bi-ortogonal 9/7 de la DWT a la imagen. Esta versión se distingue porque aplica los coeficientes de la Tabla 9 a los filtros.
3. Se cuantiza el resultado de calcular la DWT.
4. Se divide la imagen en bloques de 64x64 y cada uno de ellos se codifica utilizando el algoritmo llamado *Embedded Block Coding with Optimal Truncation* (EBCOT). Este algoritmo realiza una compresión del bloque por planos de bit, es decir, empieza comprimiendo el bit más significativo de todos los coeficientes, luego continúa por el siguiente bit, etc., hasta llegar al plano menos significativo.

5. Se reorganizan las distintas cadenas de bits obtenidas para lograr la escalabilidad del algoritmo. Así, mediante un algoritmo de optimización basado en multiplicadores de Lagrange se logra que, al indicar un tamaño de archivo, JPEG2000 lo comprime exactamente a ese mismo tamaño. Esto no ocurre con JPEG ya que no es capaz de conocer a priori el tamaño con el que se comprimirá el archivo.

k	Paso bajo	Paso Alto
0	+ 0.602949018236360	+ 1.115087052457000
+1, -1	+ 0.266864118442875	− 0.591271763114250
+2, -2	− 0.078223266528990	− 0.057543526228500
+3, -3	− 0.016864118442875	+ 0.091271763114250
+4, -4	+ 0.026748757410810	-

Tabla 9. Comportamiento asintótico

2.2.2.9 WebP

WebP es un formato de compresión desarrollado por Google que permite tanto la compresión con pérdidas como sin pérdidas.

2.2.2.9.1 Webp con pérdidas

La compresión WebP con pérdidas está basada en la predicción de bloques por lo que, en primer lugar, divide la imagen en segmentos más pequeños denominados macrobloques. Así, cada uno de los macrobloques en los que se ha dividido la imagen se predicen utilizando los bloques que ya se habían procesado. Además, existen diferentes esquemas de predicción los cuales se pueden observar en la Figura 16:

- Horizontal: rellena cada columna del bloque con una copia de la columna izquierda I.
- Vertical: rellena cada fila del bloque con una copia de la fila superior S

- DC (sólo un color): rellena el bloque con un solo valor de color, utilizando la media del valor de los píxeles en la fila superior S y la columna izquierda I
- *True Motion*: Además de la fila S y la columna I, utiliza el píxel P que se encuentra en la esquina superior izquierda del bloque para la predicción. Las diferencias entre el valor de las componentes de color horizontales de la fila S (empezando en el píxel P) se propagan y se utiliza el valor de los píxeles de la columna I para empezar cada fila. Es decir, el primer píxel de cada fila del bloque será el de la columna I y a este valor se le va sumando (o restando) la misma diferencia de luminancia que tiene lugar entre los píxeles de la fila S.
- Otros: modelos similares a la predicción horizontal y vertical que se pueden observar en la imagen.

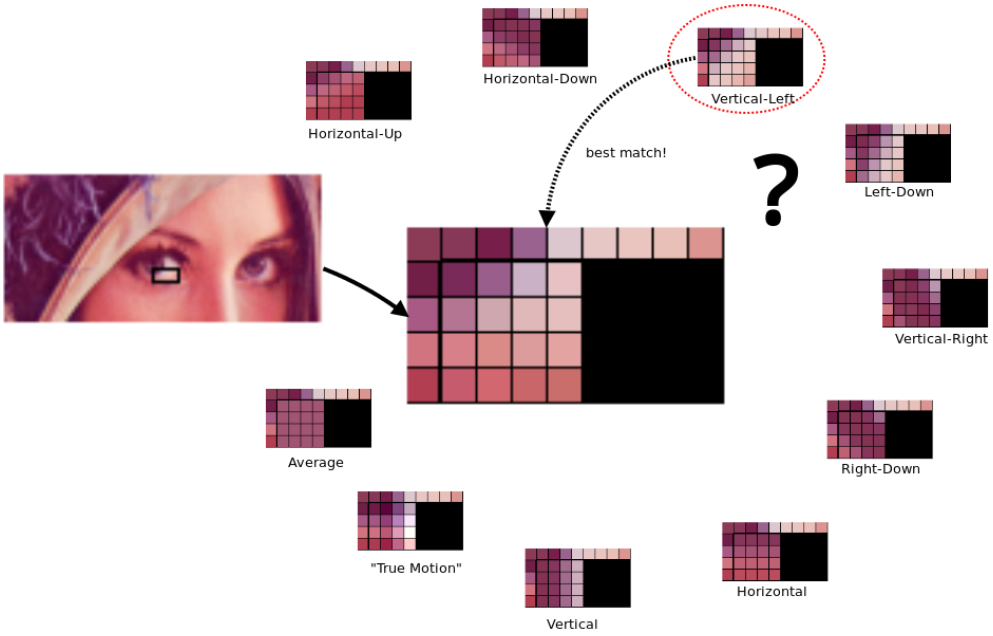


Figura 16. Esquemas de predicción.

Una vez que se encuentra la mejor predicción, ésta se resta del valor original del bloque de forma que únicamente se envía esta diferencia residual comprimida. Por otro lado, los residuos contienen muchos ceros ya que habrá habido aciertos en la predicción con respecto al valor original por lo que la compresión es aún más eficiente. Finalmente, se cuantifican estos valores y se transforman a una cadena de bits. Es en

el proceso de cuantificación en el que se producen las pérdidas de este tipo de compresión. La Figura 17 muestra el diagrama de la compresión con pérdida WebP.

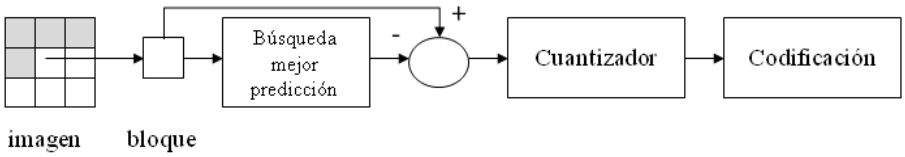


Figura 17. Compresión con pérdida WebP.

2.2.2.9.2 WebP sin pérdidas

La codificación WebP sin pérdidas se basa en la transformación de la imagen utilizando diversas técnicas diferentes:

- Predicción de píxeles: para reducir la entropía, se realiza una predicción del píxel basándose en los píxeles vecinos puesto que existe correlación entre ellos. Existen diferentes métodos de predicción del píxel pero las que más se utilizan son: izquierda, superior, izquierda-superior (mezcla de ambos) y derecha-superior.
- Transformada de color: el propósito es decorrelar los colores verde (V), rojo (R), y azul (A) en cada píxel. Esta transformada deja el color verde tal y como es, transforma el rojo según el verde y el azul según el verde y, posteriormente, según el rojo.
- Transformada de eliminación de verde: permite eliminar el color verde de las componentes de color rojo y de azul para cada píxel
- Transformada de color indexado: si el valor de la componente de color coincide en muchos píeles, es más eficiente crear un array de índice de color y reemplazar el valor del píxel por los índices del array. De esta manera, esta transformada calcula el número de valores ARGB en la imagen y si se encuentran por debajo de un umbral, se realiza la transformación.
- Caché de color: WebP utiliza la predicción para calcular el color de un determinado píxel. Sin embargo, puede ocurrir que no se encuentre una predicción adecuada por lo que existe una caché de color con diferentes valores que se utiliza en estos casos.
- Codificación: se utiliza una variante de *deflate algorith*, también utilizado en la compresión PNG.

2.2.2.10 Algoritmos basados en fractales

La compresión fractal es un método de compresión con pérdidas sujeto a una patente comercial perteneciente a la compañía *Iterated Systems*. Las imágenes se almacenan como una colección de transformadas IFS (*Iterated Function Systems*). Estas transformadas son construcciones matemáticas utilizadas para representar de manera simple conjuntos de fractales que presentan autosimilaridad. Este concepto se define en matemática como la propiedad de un objeto en el que todo es exacta o aproximadamente similar a una parte de sí mismo. Dependiendo del conjunto IFS utilizado, la tasa de compresión varía.

En el proceso de codificación fractal, una imagen se subdivide en una determinada geometría (un cuadrado, por ejemplo) que se subdivide de nuevo en cuadrados, buscando autosimilitudes entre unos y otros y almacenando algunos de los cuadrados más pequeños. A la hora de decodificar, se replican aquellos cuadrados que presentaban similitudes.

Existen varias consideraciones a tener en cuenta a la hora de utilizar este método:

- El proceso de Compresión/Descompresión es acentuadamente asimétrico. De esta manera, coste computacional de la compresión es muy elevado debido al procesamiento para encontrar autosimilitudes.

- La compresión fractal genera problemas con muchas imágenes reales. No esperamos, por ejemplo, que la imagen de un gato presente pequeños gatitos distorsionados sobre sí mismo. Para solventarlo, se subdivide aún más la imagen mediante una partición y para cada región resultante se busca otra región similar a la primera bajo las transformaciones IFS lo cual supone un procedimiento costoso.

- El método no supone una mejora significativa con respecto a sistemas anteriores en los ratios de compresión comunes (hasta 50:1). De esta manera, la compresión fractal obtiene peores resultados que los que se obtienen con la compresión basada en DCT (JPEG) salvo en compresiones más agresivas en las que puede llegar a ofrecer resultados mejores.

Debido a estas consideraciones, las imágenes fractales se utilizan para realizar imágenes artísticas y para representar formulaciones matemáticas.

2.2.2.11 Algoritmos basados en Redes Neuronales Artificiales (RNA)

Las redes neuronales artificiales (RNA) son una propuesta tecnológica para el aprendizaje y procesamiento automático de datos. Están inspiradas en la forma en que funciona el sistema nervioso central de los seres vivos: un sistema de neuronas interconectadas que trabajan entre sí para generar un estímulo de salida. En el caso de las redes neuronales artificiales, se conoce como neuronas a una serie de elementos de cálculo no lineales que se interconectan como una red con el fin de producir un resultado al insertarles unos datos de entrada. Tal y como se muestra en la Figura 18, la arquitectura de dicha red se divide en capas formadas por neuronas que se encuentran interconectadas con neuronas de otras capas. Para que la red funcione correctamente es necesario llevar a cabo un proceso de aprendizaje o entrenamiento con el objetivo de fijar los parámetros de cálculo de la red. Para ello, se utiliza un conjunto de datos denominado conjunto de entrenamiento de forma que la red fija los parámetros de las diferentes neuronas. De esta manera, si se escoge adecuadamente el conjunto de entrenamiento, es posible obtener resultados válidos para datos de entrada diferentes al conjunto de entrenamiento.

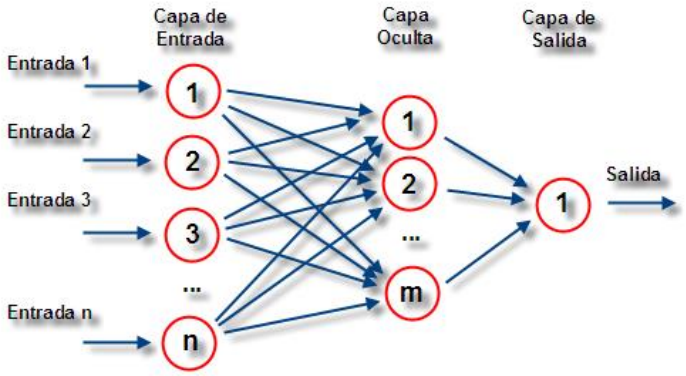


Figura 18. Capas de una red neuronal

Por otro lado, las imágenes están formadas por un conjunto de píxeles. Cada uno de ellos contiene la información acerca del brillo en ese punto de la imagen. Por tanto, se puede entender la imagen como una matriz numérica en la que cada elemento representa un determinado color. De esta manera, es lógico pensar que se puede introducir la matriz de una imagen en una red neuronal para que la aprenda y, así, llegar a comprimir la imagen. Así, el objetivo es almacenar la matriz de la imagen sin necesidad de mantenerla tal y como es sino encontrar una forma de recordar y poder reproducir dicha matriz, disminuyendo así el tamaño de la imagen. De esta manera, se busca un agente que recuerde la matriz sin necesidad de almacenarla escrita en algún lugar.

La compresión mediante redes neuronales presenta algunos problemas. Uno de ellos radica en que la búsqueda del agente capaz de recordar la imagen es lenta y costosa ya que, para cada imagen, es necesario realizar el proceso de entrenamiento de la red. De esta manera, el uso comercial de esta tecnología es aún impensable puesto que un usuario de una cámara digital no puede esperar durante horas a que la imagen se procese, codifique y almacene en la memoria.

Otro de los problemas se basa en la elección de la arquitectura de la red neuronal puesto que la arquitectura capas-neuronas nunca es la misma. Es más, es posible que para cada imagen se necesite una arquitectura distinta por lo que habría que realizar un proceso de ensayo y error costoso en tiempo para poder fijar la arquitectura de cada imagen. No obstante, este problema de arquitectura, ocurre, sobre todo, si se usan redes neuronales multicapa de algoritmo backpropagation mientras que con otro tipo de redes el problema es menos severo.

Existe una prueba piloto de compresión con redes neuronales [2]. La Figura 19 muestra la imagen original utilizada en los experimentos, de 800x600 píxeles y con un tamaño inicial en disco de 1,37MB.

Figura 19. Imagen original

Tras varios estudios, la red neuronal se construyó con una estructura de cuatro capas 1-100-50-1 (es decir, una neurona en la primera capa, 100 neuronas en la segunda, 50 en la tercera y 1 en la cuarta). Esta red cuenta con una cantidad de 100+100*50+50 = 5150 interconexiones cada una de las cuales tendrá sus pesos asignados y 1+100+50+1 =152 neuronas. Es decir, para almacenar esta red en un ordenador

necesitamos 5302 números reales o *float*. Sin embargo, si almacenásemos la matriz de la imagen serían necesarios 800x600 = 480 000 números por lo que la red, pese a su gran tamaño, sigue ocupando bastante menos que la matriz de la imagen. Tras el entrenamiento de la red y su compresión mediante la red neuronal se obtuvo el resultado de la Figura 20.

Figura 20. Imagen resultado de la compresión con RNA

Por otro lado, en la Tabla 10 se observa una comparativa en cuanto a tamaño final de la imagen y tiempo de compresión utilizando redes neuronales y métodos de compresión tales como Zip y Rar. Como se puede observar según los datos de la Tabla 10 y la imagen de la Figura 20, la relación compresión-calidad es realmente buena. Sin embargo, el tiempo de cómputo es enorme lo que hace a este sistema totalmente inificiente.

Método utilizado	**Tiempo de compresión**	**Tamaño final de la imagen**
RNA	80 min	32 KB
Zip	1 ns	432 KB
Rar	1 ns	416 KB

Tabla 10. Tabla comparativa

2.3 Algoritmos de interpolación de imagen

Los algoritmos de interpolación se utilizan, por un lado, para obtener una imagen de tamaño superior a la original rellenando los nuevos píxeles con la información resultante de aplicar el algoritmo. Por otro lado, se utilizan para la reconstrucción de imágenes a las que les falta información porque se ha eliminado en el proceso de codificación. Dicha reconstrucción es posible debido a que existe una alta correlación entre los píxeles de la imagen.

En cuanto a los diferentes algoritmos, existen algunos muy sencillos, de complejidad computacional baja, que logran resultados aceptables en calidad pero peores que otros algoritmos más complejos. Así, el algoritmo elegido para la interpolación debe mantener un compromiso entre la calidad de la imagen final resultante y los tiempos de ejecución necesarios. A continuación, se explican algunos de estos algoritmos y se aplican a la imagen ejemplo de la Figura 21 para poder observar el resultado que producen.

Figura 21. Imagen ejemplo

2.3.1 Interpolación por vecino cercano

Es el algoritmo más simple y el que requiere menor tiempo de procesamiento. Simplemente considera el píxel más cercano al punto (x,y) interpolado y copia el valor de la componente de color. Es decir, agranda el píxel. En la Figura X vemos un ejemplo de cómo se realiza dicha interpolación, el valor del píxel se copia a los píxeles más cercanos.

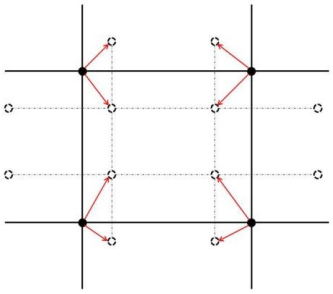


Figura 22. Interpolación por vecino cercano

Además, en la Figura 23 vemos el resultado de aplicar el algoritmo a la imagen ejemplo.

Figura 23. Resultado interpolación por vecino cercano

2.3.2 Interpolación bilineal

Se trata de una mejora de la interpolación lineal. De esta manera, se asigna al píxel un valor medio ponderado de los cuatro píxels que lo rodean. Estos factores de ponderación vienen dados por las distancias entre el píxel y los del entorno. El valor del píxel P(x,y) se calcula con la expresión dada en Ecuación 6, todos los parámetros que aparecen se encuentran definidos en la Figura 24.

$$P(x,y) = \frac{dx \cdot dy \cdot p1 + (\Delta x - dx) \cdot dy \cdot p2 + (\Delta x - dx) \cdot (\Delta y - dy) \cdot p3 + dx \cdot (\Delta y - dy) \cdot p4}{\Delta x \cdot \Delta y} \qquad (6)$$

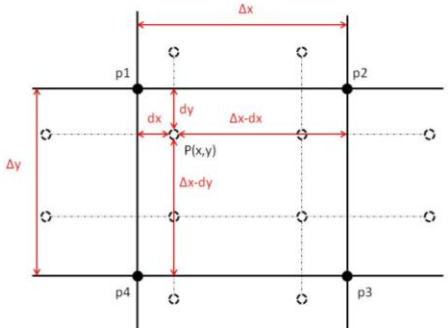


Figura 24. Interpolación bilineal

Además, en la Figura 25 vemos el resultado de aplicar el algoritmo a la imagen ejemplo.

Figura 25. Resultado interpolación bilineal

2.3.3 Interpolación bicúbica

Es el algoritmo de interpolación más utilizado ya que obtiene los mejores resultados en cuanto a calidad de la imagen resultante y el tiempo de ejecución necesario. Se trata de una mejora de la interpolación bilineal. De esta manera, en lugar de considerar los cuatro píxeles más cercanos, toma los dieciséis píxeles más cercanos al píxel (x,y) a interpolar. De esta manera, se aproxima localmente el nivel de gris en la imagen original mediante una superficie polinómica bicúbica. En la Figura 26 vemos el resultado de aplicar este algoritmo a la imagen ejemplo.

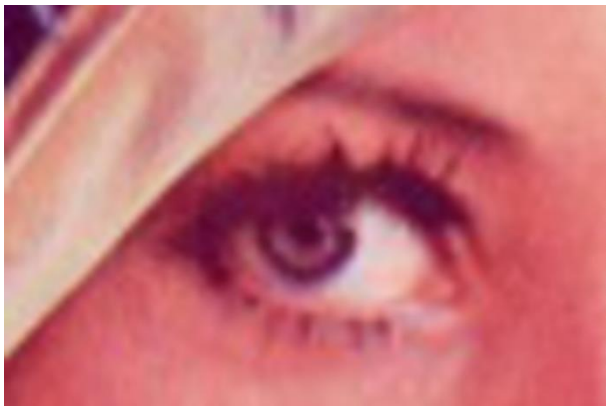


Figura 26. Resultado interpolación bicúbica

2.3.4 Otras interpolaciones

Existen otras interpolaciones tales como lanczos o HQX que ofrecen también buenos resultados en la imagen final obtenida tras el proceso de interpolación.

2.4 Codificación de vídeo

El vídeo digital apareció, por primera vez, en 1983 con el formato D-1 de Sony. Este sistema permitía almacenar la señal de vídeo de forma digital en lugar de utilizar alguna de las formas analógicas hasta el momento. Sin embargo, su alto coste dio lugar a que únicamente pudiese ser utilizado por las grandes cadenas de televisión. Finalmente, este sistema sería reemplazado por otro más barato que utilizaba datos comprimidos denominado Betacam Digital, también de Sony. Dicho formato es, aún, utilizado por productores de televisión profesionales.

No obstante, el vídeo digital para el mercado de consumo no aparecería hasta el año 1990 cuando la empresa Apple Computer lanzó al mercado un *framework* multimedia denominado QuickTime. Las primeras herramientas de creación de vídeo eran muy básicas: requerían digitalizar una fuente de vídeo analógica a un formato que pudiese interpretar el ordenador. En un principio, la calidad de estos vídeos era muy baja pero esto cambió rápidamente con la introducción de estándares de reproducción tales como MPEG-1 (1993) y MPEG-2 (1995), los cuales permiten la codificación de imágenes en movimiento y audio asociado. Actualmente, el formato MPEG-2 se continúa utilizando para los DVD y la TDT.

Desde entonces, el mercado del vídeo digital ha seguido evolucionando y creando nuevos estándares que se adaptan a los nuevos sistemas y tecnologías como, por ejemplo,

- H.263 (1996) que, en un principio, se concibió para vídeoconferencias, aunque ya no se usa.
- MPEG-4 (1998) utilizado para compresión de "vídeo en lata" (ficheros de vídeo comprimidos).
- H.264 (2003) utilizado para la grabación, compresión y distribución de contenidos audiovisuales por Internet (Youtube o productos de PayTV) .

Estos sistemas, junto con MPEG-1 y MPEG-2 se basan en la estimación de vectores de movimiento para la codificación de vídeo. Un vector de movimiento es una estimación del desplazamiento horizontal y/o vertical de cada una de las regiones que componen una imagen con respecto a uno o varios *frames* de la misma secuencia. Existen dos tipos de codificación:

- Intra-frame coding: la información se codifica referida únicamente al frame actual, sin tener en cuenta ningún tipo de codificación temporal.
- Inter-frame coding: la información enviada es relativa a otros *frames* que se han enviado anteriormente. Es decir, se envían vectores de movimiento que representan el desplazamiento de un *frame* con respecto a otro.

La codificación inter-frame es posible debido a que en imágenes en movimiento existe una redundancia temporal añadida a la redundancia espacial que poseen las imágenes. Por esta razón, no es necesario enviar toda la información de las componentes de color de cada imagen del vídeo sino que es posible enviar vectores de movimiento, es decir, las diferencias de desplazamiento que tiene lugar de un *frame* a otro. De esta manera, la imagen se divide en macrobloques y, para cada uno de ellos, se busca en una región de su alrededor en los *frames* cercanos la zona que más se asemeja. Si se encuentra, se asume que el macrobloque se ha movido. Existen dos técnicas diferentes para llevar a cabo esta estimación del movimiento:

- Estimación Forward: definimos los macrobloques en la imagen de referencia y se busca dónde se han desplazado en la imagen que se va a codificar, tal y como se muestra en la Figura 27. Como se observa, este método deja zonas de la sin definir y difíciles de codificar.

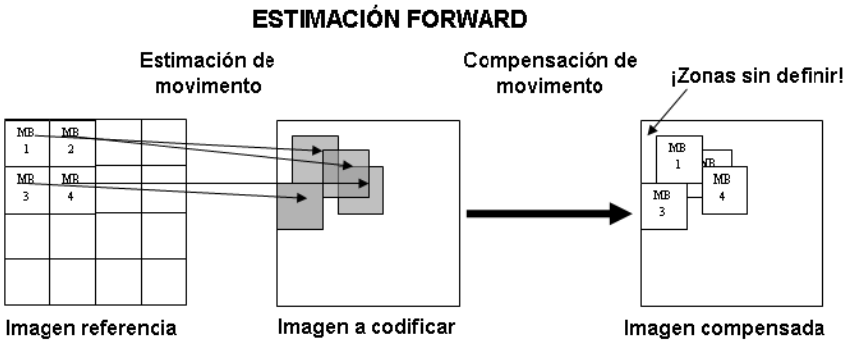


Figura 27. Estimación Forward

- Estimación Backward: define los macrobloques en la imagen a codificar y busca dónde se encontraban en la de referencia. De esta manera, el problema de la estimación Forward desaparece, tal y como se muestra en la Figura 28.

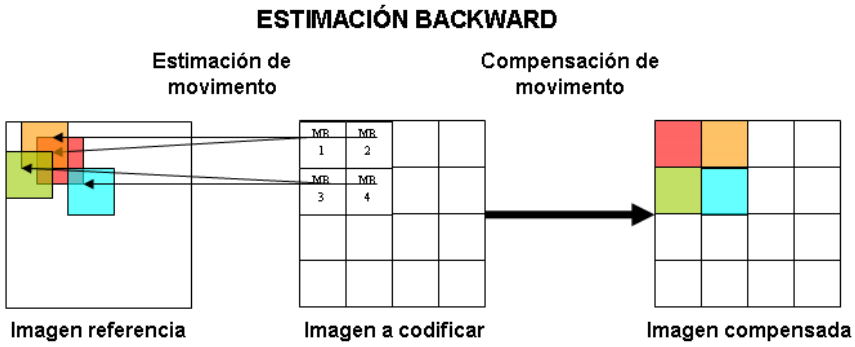


Figura 28. Estimación Backward.

Cuando se realiza la estimación del movimiento, no siempre se encuentra un macrobloque similar al de otro *frame*. De hecho, pueden haber aparecido nuevos objetos en el nuevo *frame* o haberse producido un movimiento muy brusco que no permite encontrar un macrobloque similar en el área cercana a dicho macrobloque en el *frame* de referencia. Por tanto, dependiendo del grado de similitud se realiza la codificación de una determinada manera:

1. Si existe una coincidencia total, es decir, uno de los macrobloques de la imagen actual y otro de la imagen de referencia, son idénticos y, además, se encuentran en la misma posición, se incrementa en 2 el contador secuencial de macrobloques para indicar que se salta uno. El receptor puede reconstruir esta porción de la imagen porque se encuentra en la memoria de la imagen anterior.
2. Si existe una coincidencia parcial, es decir, uno de los macrobloques de la imagen actual y otro de la imagen de referencia son idénticos pero no se encuentran en la misma posición, se introduce el vector de movimiento en la cabecera del macrobloque y la zona de datos queda vacía puesto que el decodificador obtiene los datos de la imagen de referencia.
3. Si existe una coincidencia mínima, es decir, es decir, uno de los macrobloques de la imagen actual y otro de la imagen de referencia no son iguales pero superan el umbral de coincidencia establecido como mínimo, se envía el vector de movimiento en la cabecera y en la zona de datos las diferencias existentes entre el macrobloque actual y el de referencia.
4. Si existe una coincidencia nula, es decir, uno de los macrobloques de la imagen actual y otro de la imagen de referencia son completamente diferentes, no se envía nada en la cabecera y en la zona de datos se utiliza codificación intra.

No obstante, no todos los *frames* se codifican utilizando estimación del movimiento. Como es lógico, el primer *frame* debe utilizar codificación *intra-frame* ya que no existe en memoria ninguna otra imagen con la que deducir el movimiento de los macrobloques que lo forman. Además, cada cierto número de *frames*, se vuelve a enviar uno que utiliza este tipo de codificación ya que, de esta manera, se evita que los posibles errores que hayan ocurrido en la estimación del movimiento perduren durante todo el vídeo. Estos *frames* que utilizan la codificación *intra-frame* se denominan imágenes I. Además, un archivo de vídeo estará compuesto por imágenes P, que son las que se predicen a partir de una imagen anterior en el tiempo y por imágenes B, que se calculan con referencia a una imagen anterior y otra posterior. Un ejemplo de un archivo de vídeo formado por diferentes tipos de *frame* y las referencias entre los mismos se puede observar en la Figura 29.

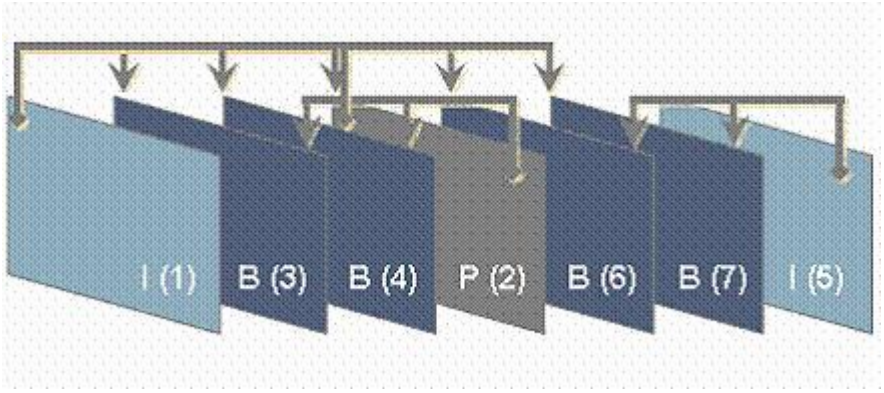


Figura 29. Imágenes IPB

Como se puede deducir, las imágenes B no pueden existir en aplicaciones de cloud gaming ya que no disponemos de un *frame* posterior al actual, se trata de un momento del tiempo que aún no ha ocurrido y no se puede retrasar la transmisión (*buffering*), tal y como se en la emisión en directo de televisión digital (TDT), puesto que esto perjudicaría a la experiencia de juego.

Finalmente, es necesario indicar que el proceso de estimación de vectores es lento y costoso puesto que hay que analizar, uno a uno, cada macrobloque de la imagen y buscar su análogo en otro *frame*. De esta manera, $\frac{2}{3}$ del tiempo total de codificación se pierde en la estimación de vectores lo que hace inviable en ocasiones los juegos online donde la latencia total no puede superar los 100 ms.

Sin embargo, en la actualidad, empresas como OnLive, Gaikai, Otoy, entre otras utilizan una variante de H.264 para la codificación de los streaming de vídeo. Esto tiene como consecuencia que los tiempos de codificación y decodificación no están suficientemente optimizados y la experiencia de usuario no es lo

suficientemente buena como para que estos servicios de vídeo interactivo vean extendido su uso. Además, debido a que estos sistemas tienen una latencia de unos 40 ms a los que además se suma la latencia de red, se hace necesario el despliegue de muchos POPs (Point Of Presence) para acercar los servidores de vídeo al usuario final con el consiguiente coste en inversión, inviable para una empresa mediana. Por esta razón, se hace necesaria la creación de un nuevo algoritmo, con menor coste computacional, que permita mejorar los tiempos de ejecución y mejorar, así, la experiencia de usuario.

Figura 30. Tiempos en un sistema de Cloud Gaming

3 Logarithmical Hopping Encoding

Las diferentes técnicas de compresión de imágenes presentes en el Estado del Arte de codificación de imagen están limitadas por: la complejidad de la imagen, calidad deseada y el coste computacional. De esta manera, algunos algoritmos permiten obtener imágenes de gran calidad pero con un alto coste computacional. Otros, sin embargo, presentan una gran simplicidad a nivel de código pero la relación entre los bits por píxel y la calidad generada no es aceptable para multitud de aplicaciones que requieren imágenes de buena calidad y de pequeño tamaño. De esta manera, nace Logarithmical Hopping Encoding (LHE). Se trata de un nuevo algoritmo para compresión de imágenes diseñado para ser computacionalmente eficiente y generar imágenes de buena calidad.

3.1 Fundamentos

La capacidad del ojo humano de diferenciar entre diferentes niveles de intensidad luminosa es de vital importancia en el tratamiento de la imagen y su posterior presentación, sobre todo si dicho tratamiento es digital y la imagen final se presenta como un conjunto discreto de niveles de intensidad, como es el caso en LHE. El rango de niveles de intensidad de luz al que se puede adaptar el ojo humano es enorme, del orden de 10^{10} niveles diferentes, desde el umbral de percepción escotópica hasta el límite de encandilamiento. Además, se han realizado experimentos en los que se demuestra que el brillo subjetivo, es decir, la intensidad que percibe el ojo humano, es una función logarítmica de la intensidad de luz que incide en el mismo. Es decir, el ojo humano solo percibe cambios de brillo si se producen variaciones logarítmicas de dicho brillo. La Figura 31 muestra la ecuación que rige la dilatación de la pupila del ojo humano, que es logarítmica.

$$\log d = 0{,}8558 - 0{,}000401\left(8{,}1 + \log B\right)^3$$

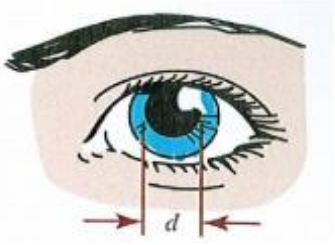


Figura 31. Dilatación de la pupila

La idea de que el ojo humano solo percibe variaciones logarítmicas de brillo, queda reflejada en la Ley de Weber-Fechner. En la Figura 32 se puede apreciar como el brillo subjetivo tiene un comportamiento logarítmico.

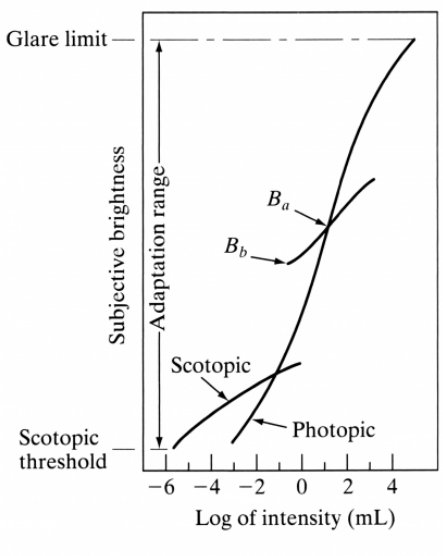


Figura 32. Brillo subjetivo en el ojo humano

Así, LHE se basa en dicha ley para codificar la imagen. Sin embargo, no la aplica a la señal luminosa sino a la predicción del error. Es decir, LHE codificará el error que se produce al realizar una predicción de la luminancia para los diferentes píxeles de la imagen. Los diferentes errores cometidos se distribuirán de forma logarítmica en el rango de luminancias posible que puede presentar un pixel.

3.2 LHE: Algoritmo básico

LHE se basa en la predicción de las señales del espacio de color YUV. Esta predicción se realiza para cada píxel en función de los anteriores. El error cometido en la predicción se codifica usando un conjunto de valores de luminancia o crominancia distribuidos logarítmicamente y denominados *hops*. En la Figura 33 se observa el diagrama de bloques de dicho algoritmo.

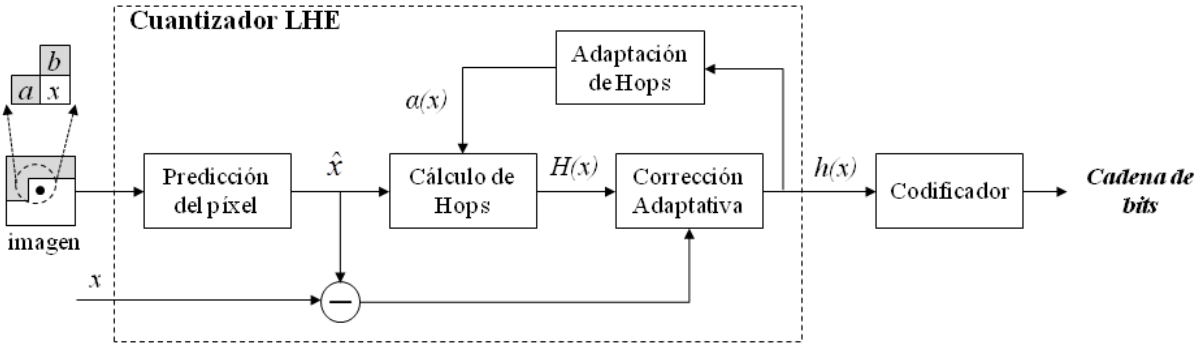


Figura 33. Diagrama LHE

A continuación, se explicará cada bloque del diagrama.

3.2.1 Predicción del píxel

LHE usa el espacio de color YUV para representar la información de cada píxel de la imagen. De esta manera, predice los valores de las componentes de cada píxel (YUV) como la mezcla del píxel superior y el píxel izquierdo, siempre que existan. La ecuación es la siguiente:

$$\hat{x} = \frac{a + b}{2} \tag{7}$$

La predicción de cada componente debe realizarse individualmente. Así, $\hat{x}$ representa el valor de la predicción de la luminancia (Y) o de la crominancia (UV). Puesto que se trata de un algoritmo experimental, otros mecanismos de predicción son posibles e, incluso, podrían mejorar los resultados que se han obtenido. Sin embargo, se usará la señal de luminancia (Y) como ejemplo de las tres componentes de color para explicar el resto de los bloques en los que se basa LHE y que se observan en la Figura 33.

3.2.2 Cálculo de *Hops*

LHE codifica el error de la predicción de las componentes YUV de un píxel teniendo en cuenta un conjunto de valores logarítmicamente distribuidos, denominados *hops*:

$$H(x)=\{h_{-N}, h_{-(N+1)}, ..., h_{-1}, h_0, h_1, ..., h_{N-1}, h_N\} \quad (8)$$

El salto nulo h_0 significa que el error asociado a la predicción es nulo o casi nulo y, por tanto, es el más pequeño que el que representa cualquier otro *hop*. El *hop* positivo y negativo más pequeño, h_1 y h_{-1}, no se asigna logarítmicamente. Por el contrario, LHE ajusta su valor automáticamente para cada píxel, dentro de un pequeño rango (entre 4 y 8), en función del píxel anterior que ha sido codificado. El parámetro que controla dicho ajuste es $\alpha(x)$ y se describe en la sección 3.2.4 Adaptación de *Hops*. Puesto que el primer píxel de la imagen no dispone de ningún píxel anterior para realizar el ajuste en función de él, se fija un valor inicial, normalmente $\alpha(0)=8$.

En la Ecuación 9 se presentan los diferentes valores que toma un conjunto de *hops H(x)* para un píxel dado.

$$h_i = \begin{cases} 0, & if\ i = 0 \\ \alpha(x), & if\ i = 1 \\ -\alpha(x), & if\ i = -1 \\ h_{i-1} \cdot ((255 - \hat{x})/k)^{1/k}, & if\ i > 1 \\ h_{i+1} \cdot (\hat{x}/k)^{1/k}, & if\ i < 1 \end{cases} \quad (9)$$

El ratio de compresión de LHE depende del número de *hops* considerado (2N+1), los *hops* pequeños, h_1 y h_{-1}, (definidos por el parámetro $\alpha(x)$) y el parámetro $k(x)$. Cuanto mayor sea N, es decir, mayor sea el tamaño del conjunto H, menor es el ratio de compresión de LHE. Los parámetros $\alpha(x)$ y $k(x)$ permiten que los valores de luminancia que presentan los diferentes *hops* del conjunto H(x) se encuentren más compactos o más dispersos para un píxel dado. El parámetro $k(x)$ debe ser mayor de 3.2 porque, de esta manera, H(x) puede cubrir todos los posibles valores de las componentes de luminancia (0-255). En el caso del algoritmo básico LHE, se calcula la constante $k(x)$ en función del parámetro $\alpha(x)$ según la Ecuación 10 donde el valor k_{ini} se inicializa con k_{ini} = 3.9. De esta manera, pretende conseguir que la compacidad de los valores del conjunto de *hops* H(x) varíe en función de $\alpha(x)$ puesto que, como se explica en la sección 3.2.4 Adaptación de *Hops*, este parámetro varía según los *hops* que han sido asignados a píxeles consecutivos de forma que se consigue un valor del parámetro *k* más eficiente y adaptado a los *hops* de la imagen.

$$k(x) = k_{ini} + (\alpha(x) - 4)x0.1275 \quad (10)$$

Para obtener buenos resultados en cuanto a calidad de la imagen usaremos, generalmente, N=4. Es decir,

$$H(x)=\{h_{-4}, h_{-3}, h_{-2}, h_{-1}, h_0, h_1, h_2, h_3, h_4\} \quad (11)$$

Como ejemplo, en la Figura 34 se puede observar los valores que toman los *hops* para N=4 e i>0 en el caso de que $\hat{x} = 128$ y h_1=4.

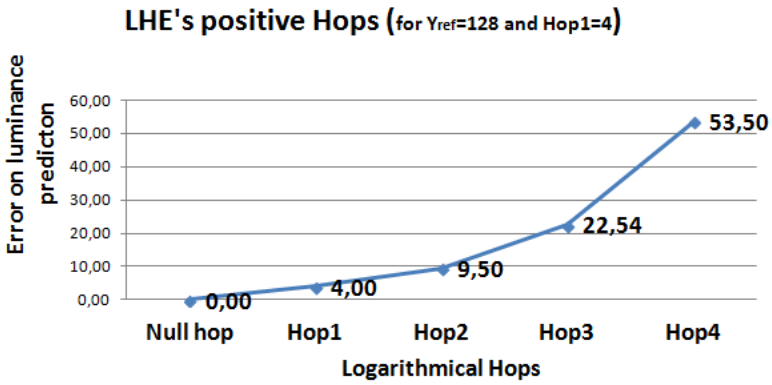


Figura 34. Valor de hops para N=4, $\hat{x} = 128$ y h_1=4.

3.2.3 Corrección Adaptativa

Los parámetros de entrada de este módulo son el conjunto de *hops H(x)*, calculado en el módulo anterior y el error asociado a la predicción realizada *e* que se calcula como:

$$e = x - \hat{x} \quad (12)$$

La salida de este módulo es el *hop h(x)* escogido y que se encuentra en el conjunto *H(x)*.

$$h(x) \in H(x) \quad (13)$$

De esta manera, el *hop h(x)* es el error cuantizado que ha sido cometido por el algoritmo LHE en la predicción de la luminancia. Este *hop h(x)* es la salida del Cuantizador LHE.

$$h(x) = arg_{h_i} \min(|h_i - e|), h_i \in H(x) \quad (14)$$

En el caso particular de que haya dos *hops* diferentes con la misma distancia al error *e*, se escoge el *hop* con el valor más pequeño. La razón de esta elección es que, a la hora de asignar los códigos a los *hops*, los códigos más cortos se asignan a los saltos más pequeños.

$$If\ \exists\left(h_j, h_k\right) \in H(x)\ \middle|\ \left|h_j - e\right| = |h_k - e| \Rightarrow h(x) = h_i\ \middle|\ \mathrm{i} = \min(|j|, |k|) \qquad (15)$$

3.2.4 Adaptación de *Hops*

Una vez que el *hop* ha sido asignado al píxel, el módulo *Adaptación de Hops* actualiza el parámetro $\alpha(x)$. Según la Ecuación 9, dicho parámetro se utiliza para calcular el conjunto de *hops H* del píxel siguiente. Los valores que puede tomar varían en un rango prefijado $[\alpha_{min}, \alpha_{max}]$, siendo α_{min}=4 y α_{max}=8. El valor para el primer píxel es $\alpha(0) = \alpha_{max}$=8 y posteriormente se ajusta su valor según las siguientes reglas:

- Si los *hops* que se han asignado a dos píxeles consecutivos son los *hops* pequeños (positivo o negativo) o el salto nulo $\{h_{-1}, h_0, h_1\}$ se decrementa el valor de $\alpha(x)$ en una unidad, siempre y cuando no se sobrepase el límite inferior α_{min}.

$$If\ \{h(x-1), h(x)\} \in \{h_{-1}, h_0, h_1\} \Longrightarrow \alpha(x) = max(h_1 - 1, \alpha_{min}) \qquad (16)$$

- Si el *hop* asignado en el módulo anterior es diferente del *hop* nulo o de los *hops* más pequeños $\{h_{-1}, h_0, h_1\}$, entonces el valor de α se resetea al valor α_{max}.

$$If h(x) \notin \{h_{-1}, h_0, h_1\} \Longrightarrow \alpha = \alpha_{max} \qquad (17)$$

3.2.5 Codificador

El codificador transforma los *hops* en una cadena de bits. Esta transformación se realiza de manera inteligente de forma que se pueda obtener la cadena más corta posible y, de esta manera, lograr un buen ratio de compresión. Un estudio realizado sobre las imágenes de dos bases de datos de imágenes diferentes que se incluyen en el apartado 5 Resultados indica que los *hops* pequeños y el *hop* nulo son asignados de forma más frecuente. Esto se debe a que, en general, los píxeles que se encuentran cercanos en una imagen suelen ser similares entre sí. De esta manera, al realizar la predicción de luminancia basada en los píxeles anterior y superior, se consigue una muy buena aproximación del valor de luminancia real del píxel por lo que el error será pequeño y se asignará alguno de los *hops* pequeños (si el error es

pequeño) o el *hop* nulo (si no ha habido error) $\{h_{-1}, h_0, h_1\}$. Además, el estudio realizado indicó lo siguiente:

- Del subconjunto de *hops* $\{h_{-1}, h_0, h_1\}$, el más frecuente es el *hop* nulo h_0 (redundancia horizontal).
- Debido a la redundancia espacial que existe en una imagen, es muy probable que el *hop* asignado a un píxel sea el mismo que el asignado al píxel superior (up) (redundancia vertical).
- Debido a la redundancia espacial que existe en una imagen, es muy probable que el *hop* asignado a un píxel sea el mismo que el asignado al píxel anterior (ant) (redundancia horizontal).

Teniendo en cuenta estas consideraciones y que nuestro objetivo es conseguir el menor número de bits por píxel, se codifica cada *hop* asignado a cada píxel de la siguiente manera:

1. Si el *hop* asignado al píxel es el *hop* nulo h_0 se asigna el código de la Tabla 11 para h_0 y se pasa al siguiente píxel.
2. Si el *hop* asignado al píxel no coincide con el *hop* nulo, se añade este *hop* a una lista de *hops* descartados y se anota un 0 en la cadena.
3. Siempre y cuando exista un píxel superior y si el *hop* asignado es igual que el *hop* superior, se asigna el código que se presenta en la Tabla 11 para up y se pasa al siguiente píxel.
4. Siempre y cuando exista un píxel superior y si el *hop* asignado al píxel actual no es igual que el *hop* del píxel superior, se comprueba si el *hop* superior está en la lista de *hops* descartados. Si no estaba, se añade a la lista y se anota un 0 en la cadena. Si lo estaba, no se hace nada.
5. Siempre y cuando exista un píxel anterior y si el *hop* asignado al píxel actual es igual que el *hop* anterior, se asigna el código que se presenta en la Tabla 11 para ant y se pasa al siguiente píxel.
6. Siempre y cuando exista un píxel anterior y si el *hop* asignado al píxel actual no es igual que el *hop* del píxel anterior, se comprueba si el *hop* superior está en la lista de *hops* descartados. Si no estaba, se añade a la lista y se anota un 0 en la cadena. Si lo estaba, no se hace nada.
7. Se codifica el *hop* del píxel actual según el código de Tabla 11. En el caso de que la longitud de los códigos asignados a los *hops* descartados sean menores que los del *hop* que estamos codificando, se recorta el código eliminando tantos ceros como *hops* descartados cumplan tal condición.

Quantized Error (*Hop*)	Code (bits)
h_0/up/ant	1
h_1	01
h_{-1}	001
h_2	0001
h_{-2}	00001
h_3	000001
h_{-3}	0000001
h_4	00000001
h_{-4}	000000001

Tabla 11. Compresión Estadística

El proceso anterior tiene una excepción que es el primer píxel de la imagen. En este caso, LHE reserva 8 bits para codificar el valor de cada componente de color de este primer píxel ya que sin una referencia de luminancia el sistema no sería capaz de realizar las futuras predicciones de Y.

3.2.6 Decodificación

El Decodificador LHE realiza unas operaciones parecidas al Cuantizador LHE, pero en orden inverso. La decodificación se hará píxel a píxel. De esta forma, puesto que para un píxel dado ya se han decodificado los anteriores, podemos conocer cuál es el *hop* anterior y el *hop* superior (necesarios para decodificar la cadena de bits) así como calcular el valor del parámetro $\alpha(x)$. A continuación, se especifican las diferentes fases del proceso de decodificación:

1. La cadena de bits se va leyendo y transformando en el valor de *hop* correspondiente para cada píxel $h(x)$. Cada píxel tiene asignado un *hop* que se corresponde con un código en bits que acaba con un 1. De esta manera, es fácil localizar donde empieza un *hop* y acaba otro por lo que se va

leyendo la cadena extrayendo, para cada píxel, el código de su *hop*. Una vez extraído el código se realizan las siguientes acciones:

a. Si la cadena está formada únicamente por un bit 1, el *hop* que se había asignado a ese píxel es el *hop* nulo h_0.
b. Si existen ceros en la cadena, se han descartado *hops*. Cada cero indicará que se ha descartado, en este orden:
    i. *Hop* nulo h_o.
    ii. *Hop* que coincide con el *hop* del píxel superior (siempre el píxel para el que estamos decodificando tenga píxel superior).
    iii. *Hop* que coincide con el *hop* del píxel anterior siempre el píxel para el que estamos decodificando tenga píxel superior).

    Si alguno de los valores anteriores coincide, entonces el cero no puede descartar el *hop* puesto que ya ha sido descartado. En este caso, el cero está descartando el *hop* siguiente de la lista anterior o es parte del código del *hop* que aparece en la Tabla 11.
c. Una vez que sabemos los *hops* que se han descartado, miramos el código restante. En este momento, sabemos que en el encoder han podido ocurrir dos cosas: o bien se ha recortado el código correspondiente a un *hop h(x)* o bien no se ha recortado. Sin embargo, conocemos las longitudes del código del *hop* anterior y del *hop* superior así como la condición de que, para recortar, los códigos de dichos *hops* tenían que ser menores que el del *hop* que estaba siendo codificado. De esta forma, se comprueban los posibles códigos (recortados y sin recortar) hasta que se descubra el *hop h(x)* que cumple las condiciones anteriores.

2. Una vez decodificada la cadena de *hops*, se recorre la imagen bloque a bloque y se realiza, en cada píxel, la misma predicción de luminancia $\hat{x}$ que se realizó en el encoder, tal y como se indica en la Ecuación 7. En el caso del primer píxel, el valor de luminancia se extrae directamente de la cadena de bits.
3. Con el valor de $\hat{x}$, α y k se calcula el conjunto de *hops H(x)* para el píxel dado según la Ecuación 9, de forma que obtenemos los posibles errores cometidos $h(x)$. En cuanto al valor de k*(x)*, se prefijará el mismo que en el encoder, es decir el dado en la Ecuación 10. Por otro lado, en cuanto a $\alpha(x)$ recordemos que, en el codificador, el valor de α para el primer píxel era $\alpha(0) = \alpha_{max} = 8$ por lo que en el decodificador se asigna el mismo valor ($\alpha(x)$ se actualizará más adelante).
4. Se calcula el valor de luminancia decodificado *x'* del píxel de la siguiente forma:

$$x' = \hat{x} + h(x) \tag{18}$$

5. Finalmente, el nuevo valor de *α(x)* es calculado siguiendo las reglas de la sección 3.2.4 "Adaptación de *Hops*".

En la Figura 35, se observa el proceso de codificación y decodificación de *hops*. El Cuantizador LHE asigna *hops* a los píxeles y, posteriormente en el decoder, estos *hops* son transformados en las luminancias correspondientes. Además, cabe destacar que la imagen que se presenta en esta figura es un icono, imposible de codificar mediante JPEG.

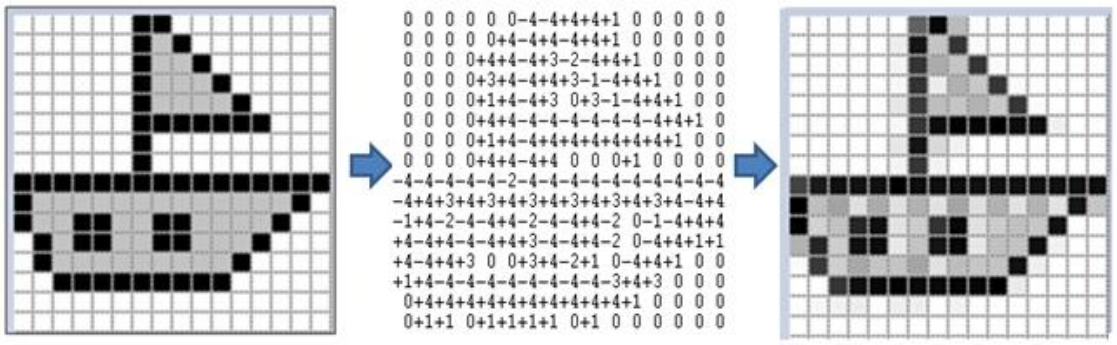


Figura 35. Ejemplo de codificación y decodificación LHE.

3.3 Relevancia Perceptual

Existe un método de codificación anterior a este documento denominado Region Of Interest (ROI). La idea principal de este método es dividir la imagen en dos zonas: regiones de interés y el fondo. Si codificamos la región de interés con mayor fidelidad que el fondo, se consigue un mayor ratio de compresión con una buena calidad subjetiva. La codificación ROI se basa en que el fondo es menos relevante para el observador que la región de interés por lo que los errores que se producen en el fondo son menos perceptibles que los que suceden en la región de interés.

De la misma manera, LHE toma la idea de que existe una determinada relevancia perceptual en las diferentes áreas de una imagen pero, en lugar de distinguir entre región de interés y fondo, LHE

generaliza el concepto de ROI y lo que hace es evaluar la relevancia perceptual de cada bloque de la imagen (8x8 pixels) y los codifica según los resultados obtenidos.

3.3.1 Evaluación de la Relevancia Perceptual. Métricas.

Intuitivamente, la relevancia perceptual de un bloque de una imagen se puede definir como la medida de la importancia de un bloque en comparación con la imagen completa, tal y como lo percibe un observador.

Tras la realización de diversos estudios durante el desarrollo del algoritmo LHE se descubrió que la luminancia cuantizada logarítmicamente, es decir, la salida del Cuantizador LHE se puede utilizar para calcular la relevancia perceptual de cada bloque. De esta manera, tras realizar una primera codificación del bloque mediante el algoritmo LHE, se procede a evaluar la relevancia perceptual. Para ello, se utilizan diferentes parámetros:

- S_{avg}: índice que indica el promedio de *hops* en un bloque (normalizado entre 0-1). Esta métrica da una medida del tamaño de los *hops*. Un valor cercano a 1 indica alta fluctuación de las componentes de luminancia y crominancia, es decir *hops* grandes. Por tanto, índica que existe una región compleja como la piel o la espuma del mar.
- S_h: índice que indica el número de cambios en los *hops* cuando se escanean los símbolos en horizontal (normalizado entre 0-1). Un valor cercano a 0 indica que el bloque tiene poca información en la dirección horizontal.
- S_v: índice que indica el número de cambios en los *hops* cuando se escanean los símbolos en vertical (normalizado entre 0-1). Un valor cercano a 0 indica que el bloque tiene poca información en la dirección horizontal.

En la Figura 36 se puede observar un ejemplo de estas métricas en una imagen en la que aparecen diferentes bloques con distintos tipos de información. El bloque de abajo a la izquierda es el que mayor fluctuación de la componente de luminancia presenta y, como consecuencia, tiene la métrica S_{avg} más cercana a 1 que el resto. En el caso del bloque de arriba a la izquierda, su métrica S_h es igual a cero porque no existe información en horizontal mientras que S_v es alta porque presenta una gran fluctuación en la dirección vertical. En bloque de arriba a la derecha presenta un degradado suave por lo que todas las métricas de Relevancia Perceptual son bajas. Por último, el bloque de abajo a la derecha presenta un borde en la dirección vertical por lo que la métrica S_h es mayor que S_v.

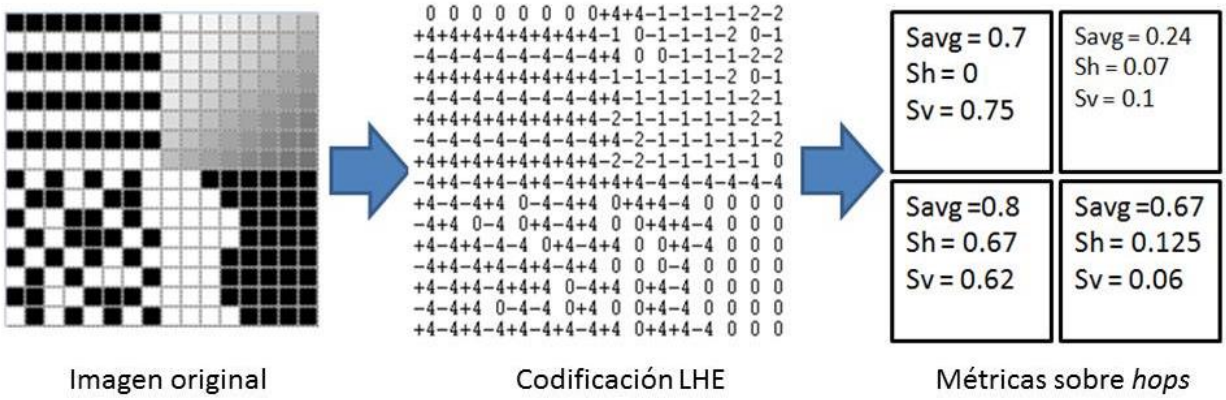


Figura 36. Métricas de Relevancia Perceptual sobre *hops*

3.4 LHE: Algoritmo mejorado

Gracias a la medida de Relevancia Perceptual, es posible mejorar el algoritmo LHE y lograr mejores ratios de compresión. La arquitectura de esta mejora es la que se muestra en la Figura 37.

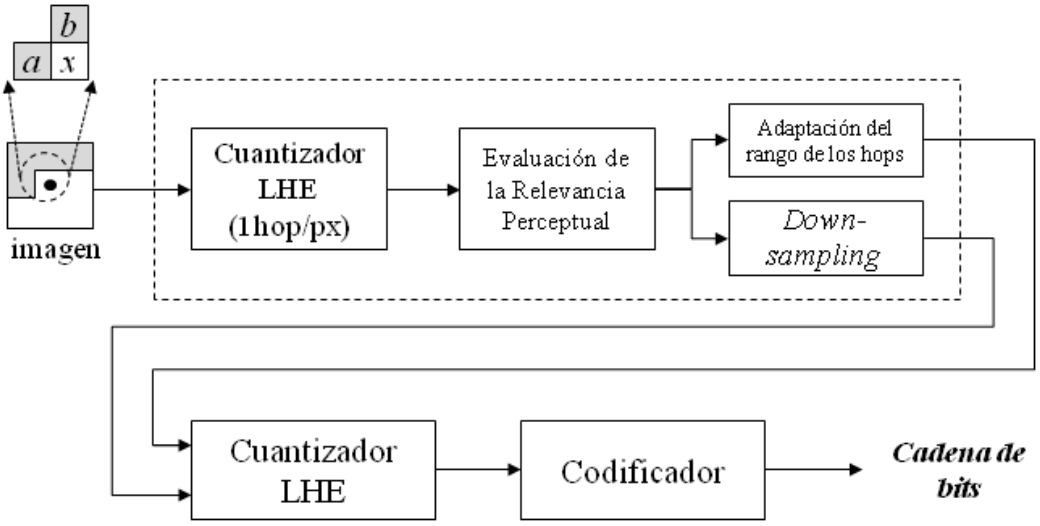


Figura 37. Diagrama LHE mejorado

Las métricas de Relevancia Perceptual se utilizan en dos nuevos módulos: un módulo dedicado a la adaptación del rango de los *hops* y otro dedicado al *downsampling* de la imagen. El objetivo del primer módulo es estimar un valor óptimo del parámetro k para cada bloque de la imagen ya que, según el valor de k, los valores que toman los diferentes *hops h(x)* del conjunto de *hops H(x)* estarán más cercanos unos de otros o más alejados. Es decir, varía el rango de error que los *hops* son capaces de abarcar. En cuanto al módulo de *downsampling*, se encarga de reducir la información que tiene que ser codificada, mejorando así el rango de compresión pero manteniendo la calidad subjetiva.

3.4.1 *Downsampling*: escalados y mallas

El módulo de *downsampling* utiliza la medida de Relevancia Perceptual para reducir la información que va a ser codificada de una manera inteligente. Para cada bloque, evalúa las métricas de relevancia perceptual definidas en la sección 3.3.1 Evaluación de la Relevancia Perceptual y, según los resultados obtenidos, se decide si se puede reducir el tamaño del bloque (escalado) de forma que se reduce la información que tiene que ser codificada.

La decisión de escalar el bloque depende de unos umbrales que se definen y son aplicados a las métricas de Relevancia Perceptual S_{avg}, S_h y S_v. Concretamente, se definen 6 umbrales: un máximo y un mínimo para cada métrica. Dependiendo del valor obtenido en cada bloque de estas métricas y su relación con los umbrales, el módulo *downsampling* estima el contenido del bloque (si hay mucha información o poca y en qué dirección) y determina si debe ser escalado.

En la Tabla 12 se resumen las reglas de detección que se aplican para identificar el tipo de información que contiene un bloque de 8x8 y la estrategia de escalado que se aplica en cada caso. También se indica el código binario que se aplica a cada tipo de escalado para que pueda llevarse a cabo, posteriormente, el proceso de decodificación.

Tipo de información	Reglas[1]	Estrategia de escalado	Código
Luminancia lisa	Savg↓ and Sh↓ and Sv↓	4x4 pixels (vertical y horizontal)	11
Degradado o detalles suaves	Savg↓ and Sh↓	8x4 (horizontal)	01
	Savg↓ and Sv↓	4x8 (vertical)	10
Detalles fuertes y borrosos, ej: pelo	Savg↑ and Sh↑	8x4 (horizontal)	01
	Savg↑ and Sv↑	4x8 (vertical)	10
Otros	No se traspasa ningún umbral	No escalado 8x8	00

[1]↓=umbral mínimo traspasado, ↑=umbral máximo traspasado

Tabla 12. Técnicas de *downsampling*

Una vez que los bloques de la imagen han sido analizados (y escalados) mediante el módulo de *downsampling*, se convierten en una entrada para el Cuantizador LHE. Como se puede observar, la principal ventaja de este proceso es que existirá una cierta cantidad de bloques de 64 píxeles que serán escalados y, por tanto, reducidos a 16 o 32 píxeles.

La elección de unos umbrales u otros permite comprimir más o menos la imagen y, por consiguiente, empeorar o mejorar la calidad de la misma, respectivamente. De esta manera, si los umbrales son muy restrictivos (máximo y mínimo cercanos a 1 y 0, respectivamente), lograremos una mayor calidad de la imagen (y menor ratio de compresión).

Con el objetivo de lograr el máximo provecho del módulo de *downsampling*, se puede utilizar un procedimiento recursivo utilizando diferentes tamaños para los bloques. De esta manera, siendo *n* el número de iteraciones de este procedimiento (denominado mallas), la imagen es dividida en macrobloques de 2^{2+n} x 2^{2+n} píxeles. Para cada uno de estos macrobloques, se calculan las métricas de Relevancia Perceptual definidas con anterioridad y se revisan las reglas de escalado. Si el macrobloque se

puede escalar (según las reglas presentes en la Tabla 12), se realizará dicho proceso. Sin embargo, en el caso de que las métricas indiquen que el macrobloque no puede ser escalado, éste se dividirá en 4 bloques y se aplicará, a cada uno de ellos, la técnica de *downsampling* indicada. Este procedimiento recursivo se aplica mientras el tamaño del bloque sea mayor o igual a 8x8 píxeles. La Figura 38 muestra un ejemplo de este procedimiento de *downsampling* recursivo: se ha aplicado a una imagen de 32x64 píxeles en la que se ha usado *n*=3 mallas.

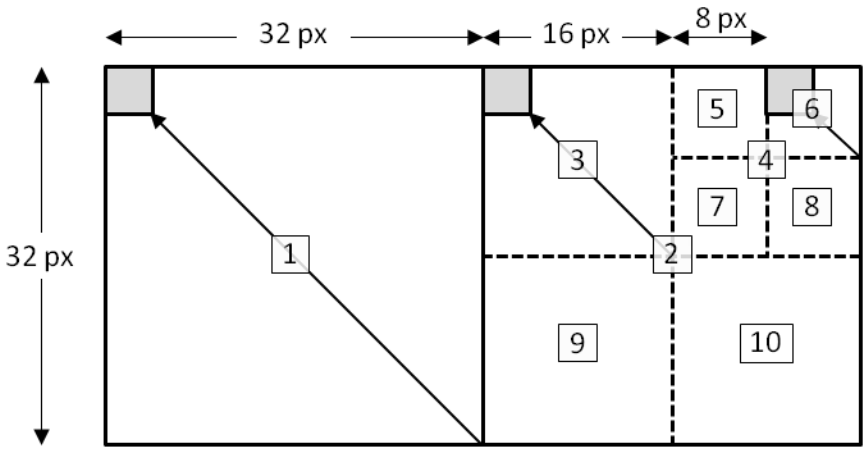


Figura 38. Procedimiento recursivo de *downsampling*

Los números que aparecen en la figura representan el orden del proceso para cada bloque y macrobloque. Como se puede observar, en primer lugar se procesa el macrobloque de 32x32, se ejecutan las métricas de Relevancia Perceptual y se descubre que puede ser escalado por lo que no es necesario ningún otro proceso sobre este bloque. A continuación, se procesa el siguiente macrobloque de 32x32. En este caso, las métricas de Relevancia Perceptual indican que no puede ser escalado por lo que se divide en bloques más pequeños de 16x16 píxeles. Después, se comienza con el primer bloque de 16x16. Este puede ser escalado por lo que se escala y se pasa al siguiente bloque. Sin embargo, el siguiente no puede ser escalado por lo que se divide en subbloques de 8x8. El proceso que se ha descrito continúa hasta que se procesa toda la imagen.

3.4.2 Adaptación del rango de los *hops*: valor de *k(x)*

Se ha visto que el ratio de compresión de LHE depende del número de *hops* considerado (2N+1), el valor de los *hops* pequeños h_1 y h_{-1} (definidos por el parámetro *α(x)*) y el parámetro k(x). De esta manera, los parámetros *α(x)* y *k(x)* son los responsables de la compacidad de los valores que toma el conjunto de hops H(x) para un píxel dado. Así, eligiendo diferentes valores de *k(x)* se consigue expandir (los valores de los *hops* están separados entre sí) o contraer (los valores de los *hops* están próximos entre sí) el rango cubierto por el conjunto H(x) de *hops* logarítmicos. En zonas de la imagen en las que existen altas fluctuaciones, un valor de *k(x)* bajo permite cubrir el rango máximo y proporciona mejores resultados. Sin embargo, en áreas de detalles suaves, un alto valor de *k(x)* permite que los valores de los *hops* se encuentren muy compactos, de forma que se obtiene una mayor precisión en pequeños cambios del valor de la luminancia que se producen en los píxeles.

De esta manera, para cada bloque, elegimos el valor k_{ini} de la Ecuación 19 para que se adapte correctamente al área de la imagen que estamos codificando. Puesto que será necesario almacenar dicho valor, los mejores resultados (relación bit por píxel-psnr) se obtienen utilizando dos valores de k_{ini} diferentes. Es decir, habrá que almacenar un bit más por cada bloque.

$$k_{ini} = \{3, 4\} \tag{19}$$

Así, un bloque de la imagen en el que la luminancia presente muchas fluctuaciones, será codificado con una $k_{ini} = 3$ mientras que si, por el contrario, los píxeles de dicho bloque presentan valores de luminancia suaves, con muchos detalles, se usará $k_{ini} = 4$.

3.4.2.1 Valor de k(x) para cada píxel

Aunque se han obtenido buenos resultados utilizando una k_{ini} diferente por cada bloque, lo ideal sería poder utilizar un valor apropiado en cada píxel, de forma que los valores del conjunto de *hops* logarítmicos se adapten mejor en cada área de la imagen. Para aplicar esta nueva lógica, es necesario modificar el diagrama de bloques de LHE que había sido presentado en la Figura 37. De esta manera, el módulo Adaptación del rango de k desaparece, ya que la adaptación se realiza para cada píxel en lugar de para cada bloque. Así, el nuevo diagrama sería el indicado en la Figura 39.

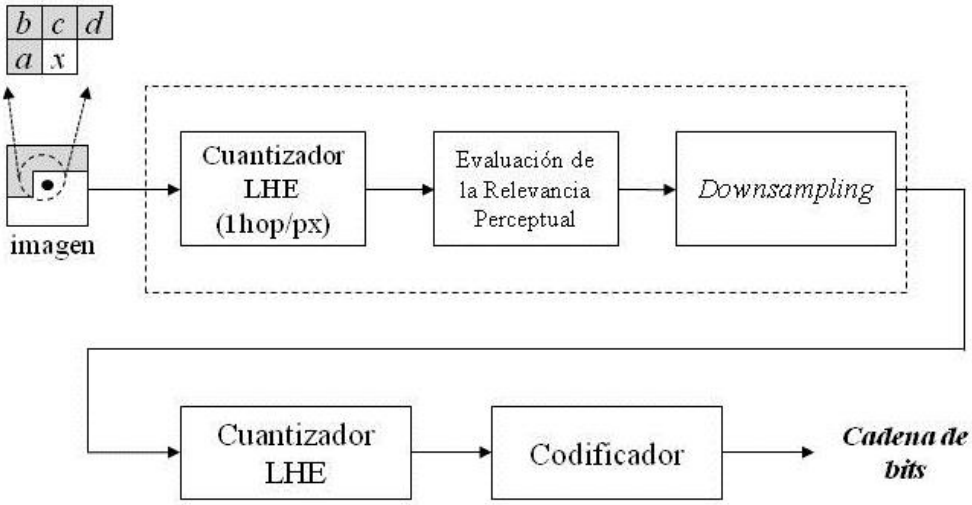


Figura 39. Diagrama LHE mejorado *k(x)* por píxel

Con esta nueva estrategia, se puede calcular k_{ini} en función de los *hops* de los píxeles a, b, c y d, que se encuentran alrededor del píxel codificado x, tal como indica la Ecuación 20. Este cálculo se realiza en el propio cuantizador LHE pero únicamente después de realizar el *downsampling* de la imagen.

$$k_{ini} = f(h(a), h(b), h(c), h(d)) \quad (20)$$

Así, para cada combinación de *hops* correspondiente a los píxeles a, b, c y d, habrá un valor óptimo de k_{ini} y, por tanto, de *k(x)* lo que permitirá minimizar el error cuando se escoja un *hop* para codificar el error cometido en la predicción de la luminancia del píxel X. Lo ideal sería encontrar una lógica para generar esa función. Sin embargo, para obtener los resultados de este proyecto utilizando esta estrategia, lo que se ha hecho es definir una tabla precalculada con valores óptimos de k_{ini}. Para ello, se han generado los valores óptimos de k_{ini} para las imágenes de la base de datos del proyecto y, por tanto, los valores óptimos para cualquier tipo de imagen ya que dicha base de datos contiene imágenes muy diversas que representan a todas las demás.

Los resultados experimentales obtenidos utilizando esta estrategia (relación bit por píxel-psnr) han sido peores que los obtenidos utilizando un valor de *k(x)* para cada bloque. Hay que tener en cuenta que LHE sigue siendo un algoritmo en desarrollo por lo que aún se puede y se debe mejorar esta estrategia en un futuro, ya que simplifica y mejora el tiempo de ejecución del algoritmo.

3.4.3 Decodificación

En el lado del decoder, el procedimiento de *downsampling* puede revertirse aplicando un algoritmo de interpolación con el objetivo de reescalar los bloques y devolverles su tamaño original. Dependiendo de la técnica seleccionada para reescalar, se conseguirán resultados diferentes en cuanto a calidad y rendimiento.

Así, en los estudios realizados con LHE se ha demostrado que los mejores resultados se obtienen utilizando interpolaciones diferentes según el tipo de bloque que se esté reescalando. De esta manera, LHE utiliza, normalmente, interpolación bicúbica por lo que debe ir, píxel a píxel, buscando los 16 píxeles cercanos y realizar la interpolación de estos valores. Sin embargo, si dos bloques consecutivos han sido escalados de forma diferente, esto es, el *downsampling* se ha realizado con dos estrategias diferentes de las que aparecen en la Tabla 12, utilizamos interpolación bilineal. Esto se debe a que, al perder resolución debido a que la reducción de información ha sido diferente, los resultados son peores si se utiliza interpolación bicúbica. Además, utilizamos interpolación por vecino cercano en los siguientes casos:

- Si el bloque es muy liso, es decir, tiene 80% de *hops* nulos (h_0).
- Si el bloque tiene un alto contraste, es decir, se mezclan *hops* nulos (h_0) y también *hops* grandes (h_{-4} y h_4) en una proporción 70%-10%, respectivamente.
- Si el bloque es muy ruidoso, es decir, existe un 40% de *hops* grandes (h_{-4} y h_4).

En la Figura 40 se muestra un ejemplo del proceso de codificación LHE. En primer lugar, se divide la imagen en bloques y se escala la iamgen según las métricas de Relevancia Perceptual obtenida. Finalmente, se reconstruye la imagen que, en este caso, ocupa 0,3 bpp y tiene un PSNR de 31,7 dB.

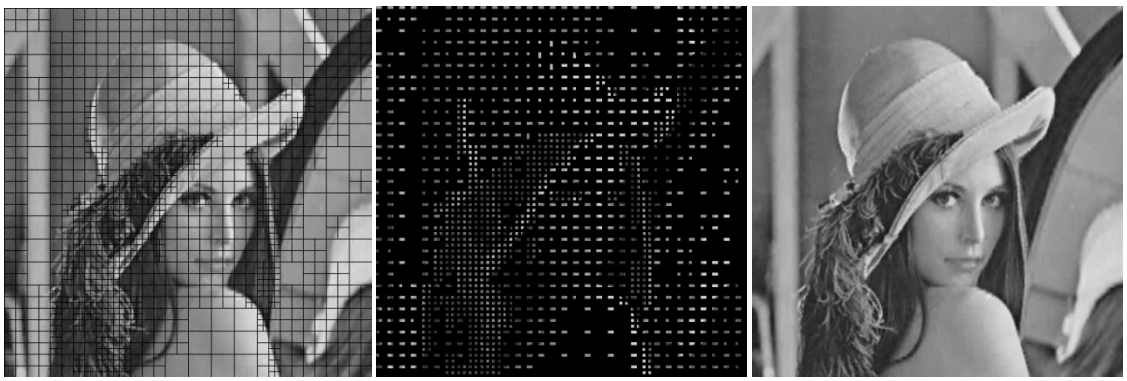


Figura 40. Procedimiento de escalado y reconstrucción de Lena

3.5 Color

Como se ha mencionado anteriormente, LHE utiliza el espacio de color YUV para representar la información del píxel. Sin embargo, en los apartados anteriores se ha tratado sobre todo la luminancia Y para explicar el funcionamiento del algoritmo básico LHE y las mejoras que se aplican sobre éste para mejorar los ratios de compresión (métricas de relevancia perceptual, mallas y escalados). En el caso de las componentes de crominancia, el algoritmo utilizado es el mismo pero existen ciertas diferencias con respecto al tratamiento de la señal de luminancia.

En primer lugar, se debe destacar que las señales de crominancia U y V son mucho más lisas que la señal de luminancia Y. Es decir, los valores que toman los píxeles en el caso de las señales U y V son muy similares entre sí. Por esta razón, el cálculo de la predicción $\hat{x}$, definido en el apartado 3.2.1 Predicción del píxel, es mucho más acertado en el caso de las señales de crominancia UV que en el caso de la señal de luminancia Y. De esta manera, el conjunto de *hops H(x)* estará definido de la misma manera que en el apartado 3.2.2 Cálculo de *Hops*: *Hops* Logarítmicos pero, dado que los errores que vamos a cometer en el cálculo de las componentes son menores, bastará con tomar N=2 para obtener buenos resultados de calidad. Es decir:

$$H(x)=\{h_{-2}, h_{-1}, h_0, h_1, h_2\} \qquad (21)$$

La reducción del conjunto de *hops H(x)* permite, además, aumentar la compresión de las señales U y V pese a que se mantiene la calidad subjetiva de la imagen final formada por las tres componentes YUV.

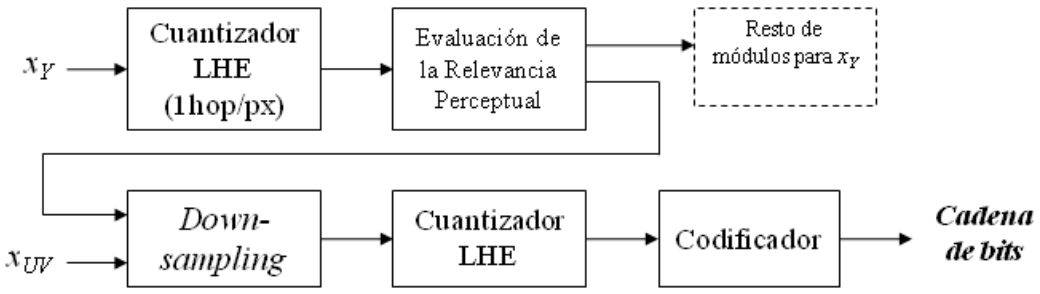


Figura 41. Diagrama LHE color

En segundo lugar, tal y como se muestra en la Figura 41, las métricas de Relevancia Perceptual definidas en la sección 3.3.1 Evaluación de la Relevancia Perceptual no se vuelven a aplicar para detectar los bloques que deben ser escalados en las señales de crominancia UV. En su lugar, el *downsampling* de las componentes de crominancia UV se realiza en todos los bloques de la imagen teniendo en cuenta el tamaño que tiene ese bloque para la luminancia Y una vez que ya se han aplicado los escalados. El objetivo es lograr una mayor compresión de las señales de crominancia. La razón que hace posible esta mayor compresión es que el ojo humano está formado por una serie de células visuales denominadas bastones (sensibles a la luz) y conos (sensibles al color). Cada ojo tiene, más o menos, 125 millones de bastones y 7 millones de conos. Es decir, el ojo humano es más sensible a los cambios de luz que a los cambios de color. Por esta razón, se pueden comprimir las imágenes de crominancia UV mucho más que la señal de luminancia Y sin que esto afecte a la calidad subjetiva de la imagen para un observador.

Tradicionalmente, se crearon diferentes modelos de *downsampling* para las señales de crominancia. Los más utilizados por estándares como JPEG son el modelo YUV 4:2:0 y el modelo YUV 4:2:2. De esta forma, LHE consigue, mediante los escalados, la misma compresión que estos modelos.

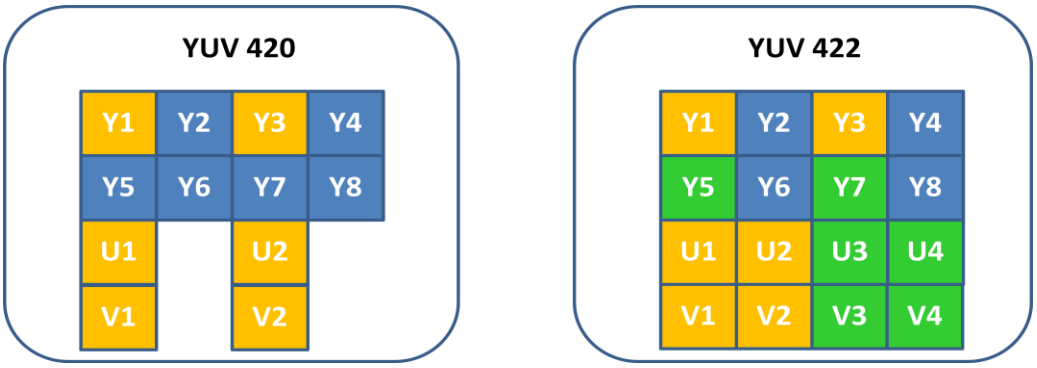


Figura 42. Modelos de crominancia

- **Modelo YUV 4:2:0**

  Tal y como se muestra en la Figura 42, el modelo YUV 4:2:0 realiza *downsampling* de las señales de crominancia. Para ello, toma, de cada dos líneas con cuatro muestras de luminancia cada una, las 4 muestras de luminancia de cada línea, 2 muestras de cada una de las componentes de crominancia UV y 0 muestras de crominancia UV de la segunda línea. De esta manera, la nueva imagen en color tendrá un tamaño de 3/2 con respecto al tamaño de la componente de luminancia.

En LHE, este *downsampling* se aplica mediante los escalados. De esta manera, tal y como se muestra en la Figura 43, para aplicar el modelo YUV 4:2:0, cada uno de los bloques de las componentes de crominancia se escalan a la mitad, tanto en la dirección horizontal como en la dirección vertical, del tamaño del bloque correspondiente a la componente de luminancia Y (una vez que se habían aplicado los escalados sobre esta componente).

- **Modelo YUV 4:2:2**

Tal y como se muestra en la Figura 42, el modelo YUV 4:2:2 realiza *downsampling* de las señales de crominancia. Para ello, toma, de cada dos líneas con cuatro muestras de luminancia cada una, las 4 muestras de luminancia de cada línea, 2 muestras de cada una de las componentes de crominancia UV y 2 muestras de crominancia UV de la segunda línea. De esta manera, la nueva imagen en color habrá doblado su tamaño con respecto al tamaño de la componente de luminancia.

En LHE, este *downsampling* se aplica mediante los escalados. De esta manera, tal y como se muestra en la Figura 43, para aplicar el modelo YUV 4:2:2, cada uno de los bloques de las componentes de crominancia se escalan a la mitad únicamente en la dirección horizontal mientras que la dirección vertical mantiene su tamaño con respecto al bloque de luminancia Y (una vez que se habían aplicado los escalados a esta componente). El hecho de que los bloques de crominancia en este modelo se escalen en la dirección horizontal en lugar de en la vertical se debe al hecho de que el ojo humano es menos sensible a los cambios que se producen en la dirección horizontal que en la vertical.

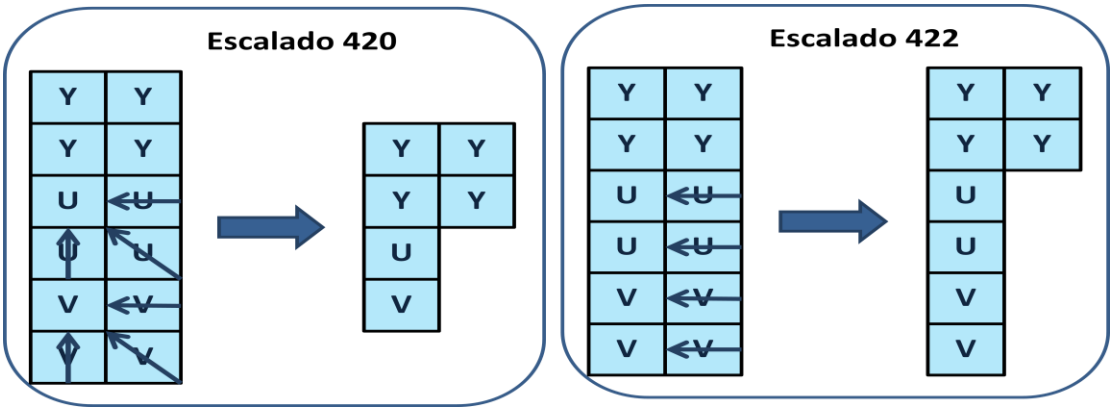


Figura 43. Escalados en crominancia

Por último, en el caso del color, no se realiza adaptación del rango de *hops*, definido en la 3.4.2 Adaptación del Rango de *Hops*. Es decir, el valor del parámetro k es siempre el mismo. Esto se debe, de nuevo, al hecho de que las componentes de crominancia son muy lisas y los errores que se cometen suelen ser pequeños por lo que no es necesaria la búsqueda de una k óptima ya que en la mayor parte de los casos se elegirá, para cada píxel, un *hop* del conjunto:

$$H(x)=\{h_{-1}, h_0, h_1\} \quad (21)$$

Según la Ecuación 21, este conjunto no utiliza el valor de k para calcular el error cuantizado que representan. Sin embargo, los *hops* h_{-1} y h_1 sí que utilizan el valor del parámetro $\alpha(x)$. Las reglas que se aplican para el cálculo de este parámetro son las mismas que las definidas para la luminancia en la sección 3.2.4 Adaptación de *hops* pero el valor con el que se inicializa dicho parámetro para el primer píxel es $\alpha(0)= \alpha_{min}=4$ ya que, en el caso de la crominancia, proporciona mejores resultados.

3.5.1 Codificación y decodificación color

La decodificación y decodificación del color se realiza de la misma manera que en la luminancia. Es decir, la estrategia que se sigue para transformar los *hops* de crominancia a bits es la misma que la que se indica en la sección 3.2.5 Codificador.

En el caso de la decodificación, se sigue la misma estrategia que la definida en la 3.2.6 Decodificación. Sin embargo, los parámetros del decoder variarán según se ha explicado con anterioridad.

4 Arquitectura Software del sistema desarrollado

El sistema está formado por varios paquetes Java que se pueden observar en la Figura 44. Cada uno de ellos contiene las clases que implementan toda la lógica del sistema. Dado que las clases son muy extensas, se procederá a resaltar las funciones más importantes de cada una de ellas para, así, comprender mejor el funcionamiento. El código de las funciones aquí mencionadas se puede encontrar en el Anexo I: Código.

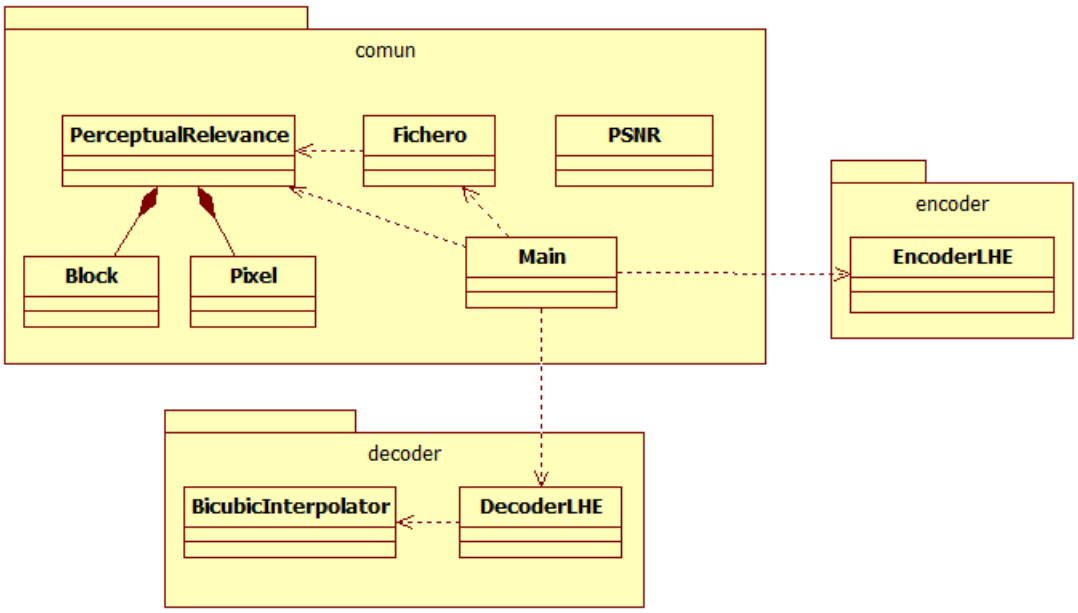


Figura 44. Diagrama de paquetes del sistema desarrollado

4.1 Package común

Este paquete contiene las clases que permiten realizar la interacción del programa con el exterior (carga de imágenes en el programa, generar ficheros…) así como las funciones que son necesarias tanto para el funcionamiento del codificador como del decodificador.

4.1.1 Clase Main

Es la clase principal. Lanza un menú que permite seleccionar si se desea codificar una imagen o bien decodificar un fichero LHE. Así mismo, permite seleccionar varias opciones. En el caso del encoder, permite seleccionar los bpp con los que se desea codificar la imagen y el modelo de crominancia (B/N, LHE 4:2:0, LHE 4:2:2).

4.1.2 Clase Fichero

Es la clase que permite transformar las imágenes en arrays para que puedan ser codificadas. Además, crea el archivo LHE mediante codificación binaria de las mallas, los *hops* y el resto de información de la imagen. De la misma manera, es capaz de decodificar dicho fichero para que, mediante la información que contiene, se pueda reconstruir la imagen.

- **BufferedImage loadImage (String pathImagen):** carga una imagen del sistema localizada en el directorio pathImagen en un objeto BufferedImage.
- **int[] imgToInt(BufferedImage img):** transforma el objeto BufferedImage en un array que contiene los valores de la componente de luminancia de cada píxel. Existen otras dos funciones análogas a esta pero que, en lugar de obtener y almacenar los valores de luminancia, permiten obtener y almacenar los valores de las componentes de crominancia U y crominancia V de cada píxel, respectivamente. Además, existen las funciones inversas a las ya nombradas, es decir, capaces de transformar los arrays con los valores de las componentes de color a su respectivo BufferedImage.
- **boolean salvarImagen(final BufferedImage imagen, final String rutaFichero, final String formato):** guarda un objeto BufferedImagen en el directorio y con el format especificados.
- **float saveLHEFile (String pathFile, int ancho, int alto, int lum, int cromU, int cromV, String mode, String malla, String image, String image_cromU, String image_cromV, boolean codificado):** genera la cadena de bits que forman el fichero LHE.
- **String mallasToBits (Vector<Integer> malla):** realiza la codificación de los diferentes tipos de escalado en cada malla transformándolos a bits.
- **String imageToBits (PerceptualRelevance pr, String modo, int[] comp_image, boolean codigo_huff):** realiza la transformación a bits de los *hops* que se han obtenido al codificar la imagen.
- **PerceptualRelevance readLHEFile (String pathFile):** junto con las funciones generaHops (permite decodificar los *hops*) y generaBlocks (a partir de la información de los tipos de escalado se obtienen los bloques en los que se dividió la imagen en el decoder), readLHEFile decodifica la cadena de bits procedente de un fichero LHE: obtiene la información de la imagen original (ancho, alto, modo de crominancia…), obtiene los tipos de escalados de cada bloque (mallas), genera los hops. Además, devuelve un objeto PerceptualRelevance que contiene la información acerca el ancho, alto de la imagen, obtenidos al decodificar el fichero y del número de mallas y la longitud mínima del bloque, parámetros prefijados en el encoder y en el decoder.

4.1.3 Clase PerceptualRelevance

Esta clase se encarga, en el encoder, de aplicar las métricas de Relevancia Perceptual a cada macrobloque para poder escalar la imagen y, así, reducir su tamaño. De esta manera, devuelve una lista con la lista de subbloques escalados que se deben pasar al codificador. Además, contiene las funciones necesarias para realizar la reconstrucción de la imagen mediante algoritmos de interpolación en el lado del decoder. Por otro lado, la clase PerceptualRelevance tiene dos subclases:

- Subclase Píxel: se utiliza para representar a cada píxel por su brillo y sus coordenadas en la imagen original y en la imagen escalada.
- Subclase Block: se utiliza para representar cada bloque por sus coordenadas originales y escaladas, la malla a la que pertenece y el tipo de escalado asignado al mismo.

A continuación, se explican las funciones y constructores más importantes de PerceptualRelevance:

- **Constructores:** existe un constructor para el encoder y otro para el decoder. Ambos permiten inicializar los parámetros básicos del sistema (ancho y alto de la imagen, número de hops, número de mallas) así como los arrays que contendrán la información del valor de las componentes de color en cada píxel y los que contendrán la información relativa a los *hops*. Por otro lado, la diferencia fundamental entre uno y otro es que, en el caso del constructor del encoder, se inicializan los valores de los umbrales de las métricas de Relevancia Perceptual mientras que el del decoder inicializa las variables necesarias para realizar la interpolación de la imagen y obtener el resultado final.
- **void THconfigCalidad(float calidad):** esta función se llama desde el constructor del encoder y permite que, dada una calidad que se pasa como parámetro, inicializar los valores de los umbrales para que el escalado de la imagen sea más o menos agresivo, en función de los bits por píxel deseados por el usuario.
- **void computeMetrics(int macrobloque):** calcula los valores de las métricas de Relevancia Perceptual del macrobloque que se le pasa por parámetro y, con esos valores y los umbrales ya inicializados, determina el tipo de escalado a aplicar en los bloques de la malla 0. Finalmente, llama a la función fillScaleType.

- **void fillScaleType(int malla, int mb):** rellena los tipos de escalado para la malla que se indica por parámetro (si se trata de la malla 0 no hace nada puesto que se ha rellenado en computeMetrics). Los tipos de escalado son los siguientes:
    - Escalado tipo 0: 4x4 píxeles
    - Escalado tipo 2: 4x8 píxeles
    - Escalado tipo 3: 8x4 píxeles
    - Escalado tipo 1: no escala, el bloque se deja igual.
- **void escaleArea(int xini,int yini,int xfin, int yfin, int xfinscaled, int yfinscaled):** calcula las luminancias del bloque escalado teniendo en cuenta las coordenadas originales y las coordenadas escaladas del mismo. La función da más peso los valores de las componentes de color de los píxeles cercanos al píxel escalado.
- **ArrayList<Block> escaleBlock(int macroblock, int malla, String mode):** actúa sobre el array de luminancias originales. Escala el bloque en la imagen dada de acuerdo con las métricas calculadas y llamando a la función escaleArea. La información referente a los bloques se añadirá al ArrayList de objetos Block, de forma que se pueda pasar esta lista al codificador. El parámetro mode debe indicar si estamos escalando señal de luminancia o de crominancia, puesto que las señales de crominancia reducen a la mitad el tamaño de escalado con respecto a las de luminancia.
- **void interpolaBlocks(boolean bilineal, int [] encoded, int[] rescaled_final, String mode):** en el decoder, interpola los píxeles de la imagen para recuperar la imagen completa a partir de la imagen escalada. Se recorren los píxeles de la imagen y, para cada píxel, se averiguan los 16 píxeles de alrededor para realizar interpolación bicúbica. En el caso de que dos bloques consecutivos se hayan escalado de forma diferente, se usará interpolación bilineal entre 4 píxeles. También se puede indicar que se desea interpolación bilineal indicándolo en el parámetro correspondiente al llamar a la función. El resultado de la imagen se guarda en el array rescaled_final.
- **void interpolaBlocksVecino(int[] encoded, int[] rescaled, String mode):** permite interpolar la imagen utilizando el método de vecino cercano. El algoritmo se encuentra implementado en la función **public void rescaleVecino(int[] rescaled, int[] encoded, int xini, int xfin_orig, int xfin_scaled, int yini, int yfin_orig, int yfin_scaled)**.

4.1.4 Clase PSNR

Contiene las funciones necesarias para evaluar el PSNR de la imagen y, de esta manera, poder ofrecer los resultados comparativos que se muestran en la sección 5 Resultados.

4.2 Package Encoder

Este paquete contiene una única clase que es la que permite realizar la codificación LHE de la imagen.

4.2.1 Clase EncoderLHE

Esta clase permite controlar el bucle de macrobloques para realizar una primera codificación LHE de los mismos y calcular las métricas de Relevancia Perceptual. Una vez que se ha obtenido la lista con los nuevos bloques escalados y sus coordenadas, la clase controla el bucle de bloques para ir codificando, uno a uno, los bloques de la lista.

- **void hazLHEMacrobloque (PerceptualRelevance pr, String mode, int[] lumin_orig, int[] crominanciaU_orig, int [] crominanciaV_orig, int ancho, int alto, int num_hops, int num_mallas, int block_len_min):** inicializa todas las variables necesarias para la codificación. Recorre los macrobloques de la imagen y para cada uno de ellos, se llama a la función que calcula los *hops*. Posteriormente, se realiza el cálculo de métricas PerceptualRelevance a partir de los *hops* calculados. Finalmente, se llama a la función hazLHEBloques.
- **void hazLHEBloques (PerceptualRelevance pr, String mode, ArrayList<Block> lista):** recorre los bloques del ArrayList y llama a la función que calcula los *hops* pero, en esta ocasión, únicamente se codificarán los bloques teniendo en cuenta las coordenadas escaladas
- **void comprime9SymbolsBloque(PerceptualRelevance pr, int[] comp_bloque, int[] comp_image, int[] int_image, int[] array_ofp, int[] escaled, int[] rescaled, int[][] rescaled_hops, int xini, int yini, int xfin_scaled, int yfin_scaled, int xfinorig, int yfinorig):** calcula los *hops* de cada bloque a partir del array de luminancias int_image y pasándole por parámetro a la función las coordenadas del bloque. Almacena los *hops* del bloque en el array comp_bloque (se crea un nuevo array para cada bloque) y los de la imagen completa en el array comp_image (se guardan los *hops* de toda la imagen escalada). Además, se va almacenando en array_ofp el valor del parámetro $\alpha(x)$ para poder ir recuperando el valor en cada momento.

4.3 Package Decoder

Este paquete contiene las clases necesarias para realizar la decodificación de los *hops* y la reconstrucción de la imagen mediante los algoritmos de interpolación.

4.3.1 Clase DecoderLHE

Clase que permite, a partir de los *hops* y la lista de bloques decodificada, reconstruir la imagen escalada. Una vez obtenida, se aplica alguno los algoritmos de interpolación de la clase PerceptualRelevance que permiten reconstruir la imagen por completo.

- **void decodeBloquesLHE (PerceptualRelevance pr, int ancho, int alto, String mode, int[] comp_image, int[] comp_cromU, int[] comp_cromV, ArrayList<Block> lista):** utilizando la lista de bloques y los *hops* que estaban almacenados en el fichero LHE y que ha sido decodificados en la clase Fichero, se reconstruye la imagen. Para ello, en primer lugar, se inicializan todas las variables necesarias para el proceso. Posteriormente, se extraen los *hops* de cada bloque y, con ellos, se rellena el array que contiene el valor de $\alpha(x)$. Posteriormente, se llama a la función decode9SymbolsBloque que obtendrá los valores de la componente de color en cada píxel. Una vez decodificada toda la imagen, se aplican las funciones de interpolación que se encuentran en la clase PerceptualRelevance.
- **void decode9SymbolsBloque(PerceptualRelevance pr, int ci, int[] array_ofp,int[] int_image, int[] comp_bloque, int[] comp_image, int [] rescaled, int[][] rescaled_hops, int xini, int yini, int xfin_scaled, int yfin_scaled, int xfinorig, int yfinorig, boolean crominancia):** obtiene los valores de la componente de color para cada píxel de la imagen escalada.

4.3.2 Clase BicubicInterpolator

Clase que permite realizar la función de interpolación bicúbica.

5 Resultados

Las pruebas con LHE se han realizado utilizando el algoritmo para comprimir un total de 68 imágenes a diferentes bit rates. Las librerías se han descargado de Internet [4][5] y contienen imágenes cuyas características las hacen aptas para mostrar los resultados que ofrecería el algoritmo al ser aplicado sobre cualquier imagen. A continuación, se presenta el contenido de cada una de las bibliotecas descargadas:

Kodak Lossless True Color Image Suite

Kodim01

Kodim02

Kodim03

Kodim04

Kodim05

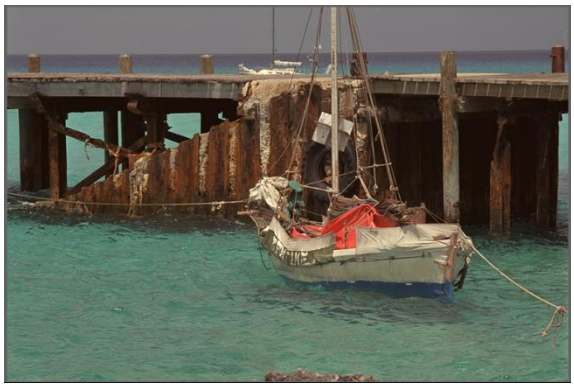
Kodim06

Kodim07

Kodim08

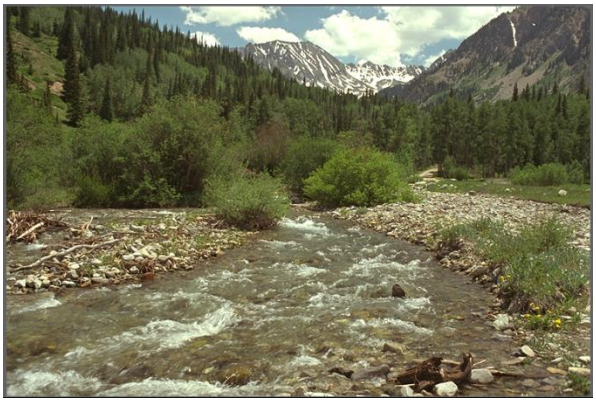
Kodim09

Kodim10



Kodim11

Kodim12



Kodim 13

Kodim14

Kodim15

Kodim16

Kodim17

Kodim18

Kodim19

Kodim20

Kodim21

Kodim22

Kodim23

Kodim24

The USC-SIPI Image Database Volume 3: Miscellanius

4.**1.01**

4.1.02

4.1.03

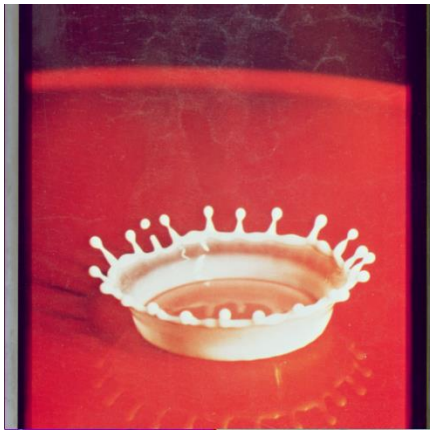
4.1.04

4.1.05

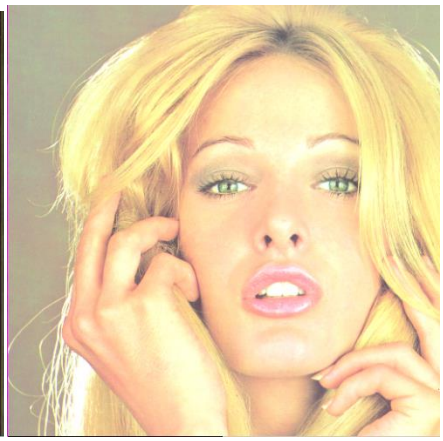
4.1.06

4.1.07

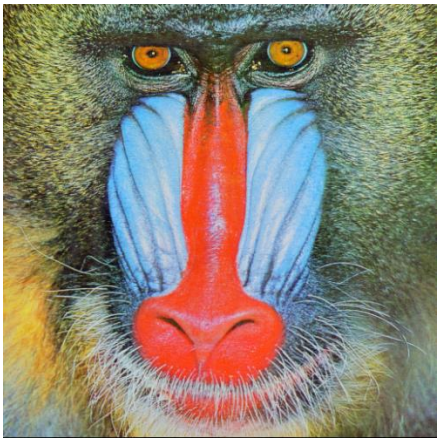
4. 1.08



4.2.01



4.2.02



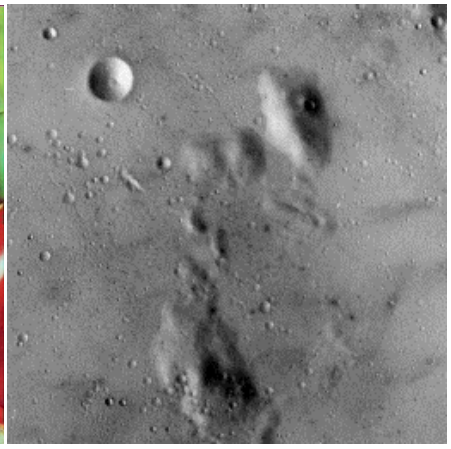
4.2.03

4.2.04

4.2.05

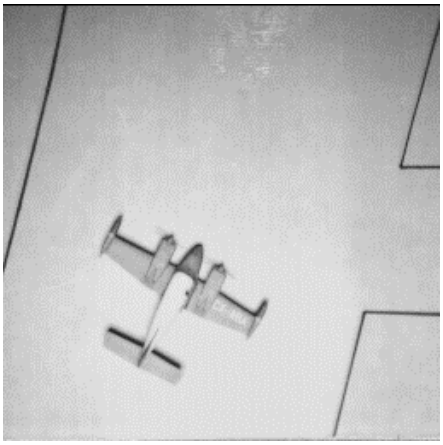
4.2.06

4.2.07

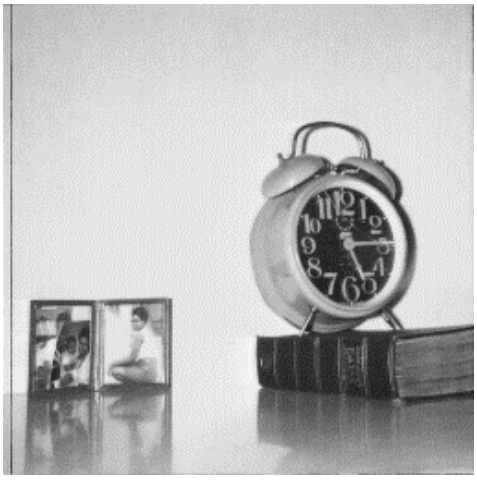


5.1.09

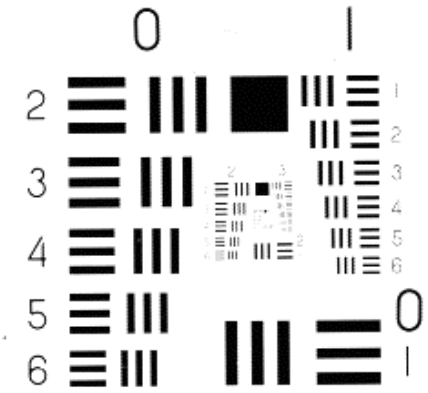
5.1.10



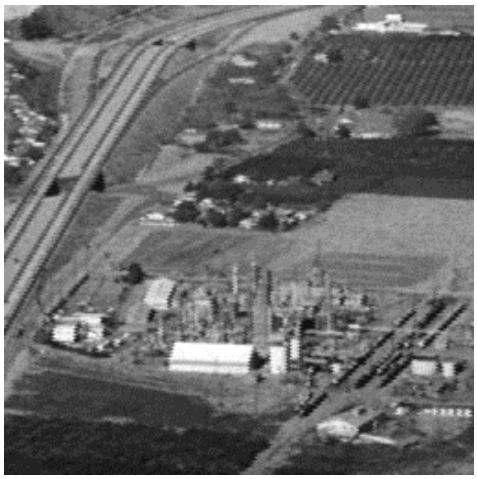
5.1.11



5.1.12



5. 1.13



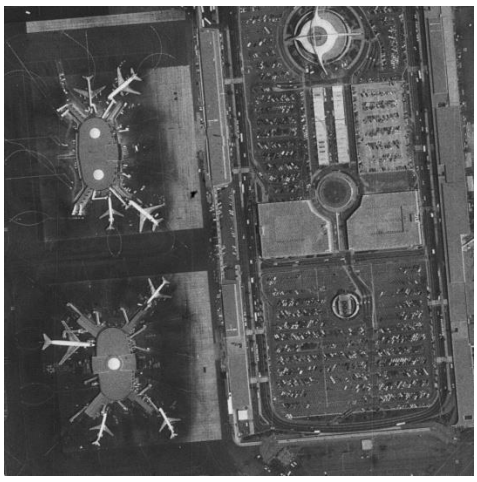
5.**1.14**

5.2.08

5.2.09

5.2.10

5.3.01



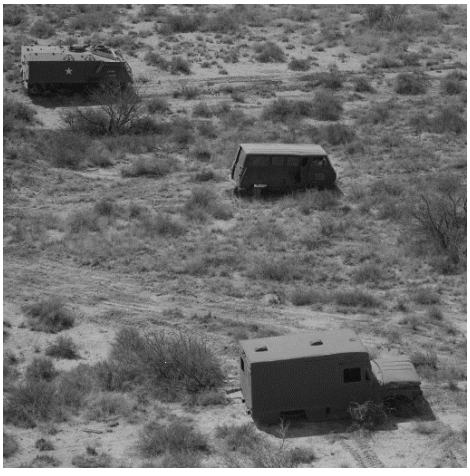
5.3.02

7.1.01

7.1.02

7.1.03

7.1.04



7.1.05

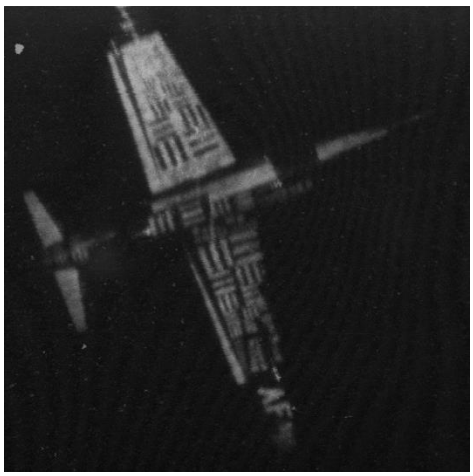
7.1.06

7.1.07

7.1.08

7.1.09

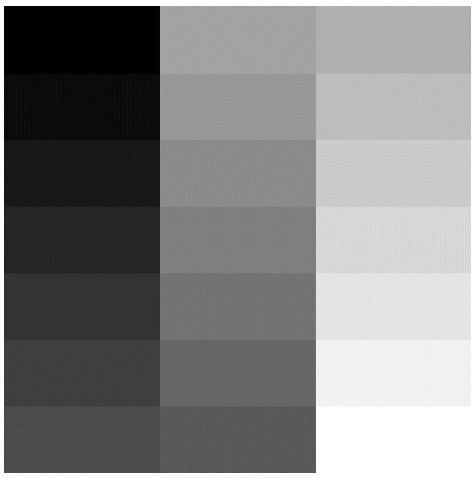
7.1.10



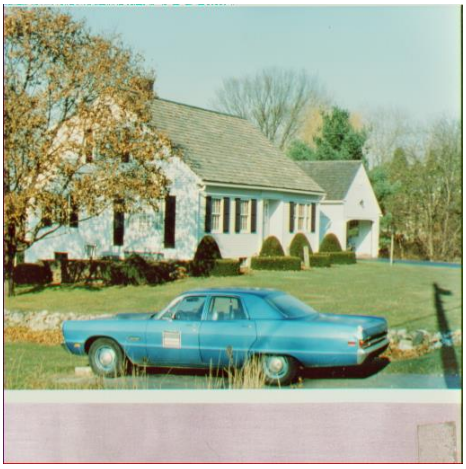
7.2.01

boat

Elaine

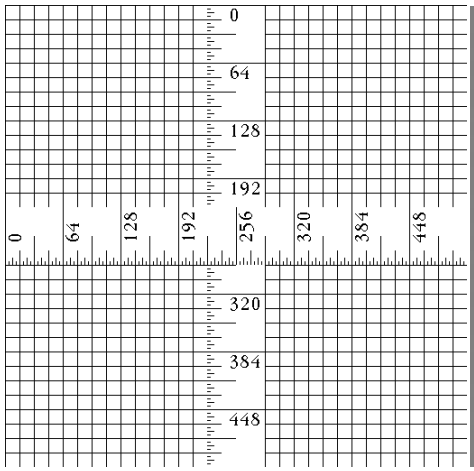


Gray21

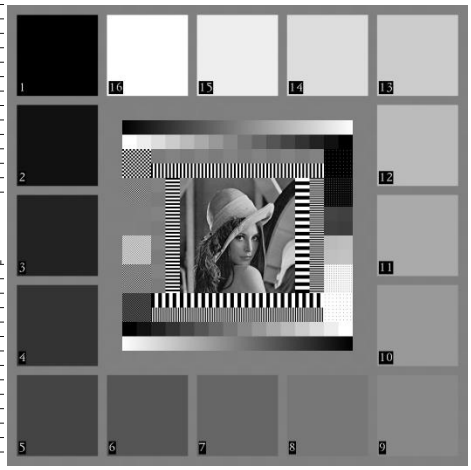


House

Numbers



Ruler



TestPat

Los experimentos consisten en aplicar el algoritmo LHE a las imágenes anteriores utilizando diferentes tasas de bit, comprendidas entre 0,05 bpp y 1,2 bpp. De esta forma, tras realizar la codificación y decodificación LHE, se obtiene una nueva imagen, parecida a la original, pero que, por tratarse de un algoritmo con pérdidas, tiene una calidad peor. Por esta razón, se comparan la imagen original y la que se ha obtenido tras la decodificación con el objetivo de hallar el PSNR. El procedimiento para realizar este cálculo se encuentra descrito en el Apartado 2.2.2 Lossy vs Lossless.

En las páginas siguientes se muestran los resultados obtenidos en los experimentos. En primer lugar, se mostrarán las tablas con el PSNR de cada imagen según la tasa de bit escogida para la compresión con LHE. En dichas tablas, las tasas de bit vienen representadas en bits por píxel (bpp) mientras que el resultado de PSNR se expresa en decibelios (dB). Se ha de tener en cuenta que estos números son el resultado de aplicar el algoritmo sobre la señal de luminancia de las imágenes anteriormente expuestas es decir, dichas imágenes en blanco y negro. Además, se mostrarán los resultados de comprimir las mismas imágenes con JPEG y se realizará una comparativa entre ambas compresiones. Es necesario comentar que, aunque en ocasiones el resultado en PSNR de la imagen con LHE sea inferior al que se logra con JPEG, la calidad subjetiva de la imagen es mucho mejor con LHE puesto que los errores son cometidos en las zonas menos relevantes para el observador. Como ejemplo, se ha codificado la imagen que aparece en la Figura 45 a 0,4 bpp con ambas codificaciones. De esta manera, mientras que con JPEG se consigue un PSNR de 23,8 dB, LHE ofrece un PSNR de 22,6 dB. Sin embargo, si se amplía la imagen, se descubre que LHE codifica mejor los detalles y los bordes pese a que su PSNR sea inferior.

Figura 45. Águila. **Detalle garra JPEG** **Detalle garra LHE**

Posteriormente, se mostrarán ejemplos del resultado final de la codificación con LHE de algunas de las imágenes representativas de la librería y se mostrará su gráfica Rate-Distorsión, comparada con JPEG. Por último, se incluirán resultados de la codificación en color.

			PSNR LHE												
NOMBRE	**BIBLIOTECA**	**TAMAÑO**	**0,05**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**	**1,1**	**1,2**
kodim01	*KODAK*	768x512	20,64	20,96	21,84	23,09	23,71	24,19	24,71	25,63	26,55	27,16	27,80	28,22	28,94
kodim02	*KODAK*	768x512	27,76	29,10	29,87	30,97	31,90	32,95	33,82	34,51	36,09	36,67	37,34	38,13	38,59
kodim03	*KODAK*	768x512	27,23	29,15	30,42	32,11	34,15	35,34	36,67	37,72	38,59	39,68	40,35	41,02	41,69
kodim04	*KODAK*	512x768	26,43	27,84	29,32	30,81	31,90	33,51	34,51	35,34	36,09	36,67	37,34	38,13	38,59
kodim05	*KODAK*	512x768	19,18	19,88	21,30	22,72	23,48	24,15	24,83	25,85	26,58	27,52	28,17	28,99	29,50
kodim06	*KODAK*	768x512	22,11	22,80	24,19	24,89	25,88	26,49	27,64	28,59	29,62	30,42	30,97	31,70	32,33
kodim07	*KODAK*	768x512	24,51	24,51	27,45	29,21	30,97	32,33	33,66	34,71	36,67	37,72	38,13	38,59	39,68
kodim08	*KODAK*	768x512	17,56	18,30	19,80	21,29	21,48	22,05	22,59	23,83	24,81	25,58	26,31	26,83	27,64
kodim09	*KODAK*	512x768	24,86	26,27	29,10	30,28	32,11	34,15	35,34	36,37	36,99	37,72	38,13	38,59	39,10
kodim10	*KODAK*	512x768	24,81	26,02	28,45	30,21	32,00	33,51	34,71	35,83	36,67	37,34	38,13	38,59	39,10
kodim11	*KODAK*	768x512	23,21	24,14	25,98	26,83	27,64	28,59	29,38	30,35	31,32	32,33	32,95	33,66	34,71
kodim12	*KODAK*	768x512	26,24	27,01	28,54	30,89	32,11	32,95	34,15	35,83	36,99	37,72	38,59	39,10	39,68
kodim13	*KODAK*	768x512	19,03	19,52	20,48	21,45	22,13	22,53	22,88	23,16	24,13	24,79	25,37	25,88	26,70
kodim14	*KODAK*	768x512	22,43	23,34	25,16	25,93	26,86	27,68	28,17	29,10	30,28	30,81	31,50	32,11	32,69
kodim15	*KODAK*	768x512	25,19	25,73	28,64	30,42	31,60	32,57	33,66	34,71	35,83	36,67	37,34	38,13	38,59
kodim16	*KODAK*	768x512	26,26	26,99	28,45	28,99	29,81	31,23	32,69	33,82	34,71	35,34	36,09	36,99	37,72
kodim17	*KODAK*	512x768	24,68	26,00	28,64	29,81	31,14	32,69	34,15	34,91	35,83	36,37	37,34	37,72	38,59
kodim18	*KODAK*	512x768	22,22	23,09	24,85	25,44	26,17	27,02	28,22	29,10	29,87	30,73	31,41	32,00	32,45
kodim19	*KODAK*	512x768	22,09	23,31	25,75	26,43	27,64	29,32	30,81	31,70	32,11	32,82	33,51	34,15	34,71
kodim20	*KODAK*	768x512	24,37	25,24	27,80	29,56	31,70	33,08	34,91	35,83	36,99	37,72	39,10	39,68	40,35
kodim21	*KODAK*	768x512	22,60	23,62	25,65	26,64	27,34	28,64	29,74	30,65	31,41	32,22	33,08	34,33	35,12
kodim22	*KODAK*	768x512	25,01	25,52	27,20	28,00	28,84	30,28	30,97	31,70	32,45	32,95	33,82	34,51	35,58
kodim23	*KODAK*	768x512	27,23	30,14	32,11	34,15	35,34	38,59	39,68	40,01	40,35	41,14	41,93	42,72	43,52
kodim24	*KODAK*	768x512	21,91	21,91	24,17	24,75	25,37	26,43	27,27	28,26	29,50	30,35	31,23	31,90	32,45

			PSNR LHE												
NOMBRE	**BIBLIOTECA**	**TAMAÑO**	**0,05**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**	**1,1**	**1,2**
4.1.01	*CALIFORNIA*	256X256	26,17	26,17	29,68	31,41	32,69	34,15	34,91	36,09	36,99	37,72	38,13	39,10	39,10
4.1.02	*CALIFORNIA*	256X256	26,14	26,14	28,99	30,07	31,90	33,08	34,51	35,12	36,09	36,67	37,34	38,13	38,59
4.1.03	*CALIFORNIA*	256X256	25,25	28,35	31,23	35,83	36,99	38,13	39,10	40,35	40,35	41,14	41,93	42,72	42,25
4.1.04	*CALIFORNIA*	256X256	25,39	25,39	27,96	30,21	31,90	33,66	34,33	34,91	35,58	37,72	38,59	39,10	39,68
4.1.05	*CALIFORNIA*	256X256	23,60	24,85	27,34	28,59	30,35	31,32	31,90	31,90	32,22	32,95	37,34	38,13	38,59
4.1.06	*CALIFORNIA*	256X256	19,58	19,58	22,69	23,36	24,59	25,73	26,46	27,45	28,22	28,64	29,10	29,62	30,35
4.1.07	*CALIFORNIA*	256X256	27,64	30,57	34,71	37,72	39,68	41,14	41,46	41,77	42,09	42,41	42,73	43,04	43,36
4.1.08	*CALIFORNIA*	256X256	25,10	25,10	30,57	33,66	35,34	36,67	38,13	39,10	40,35	41,23	41,41	41,58	41,76
4.2.01	*CALIFORNIA*	512X512	27,80	30,07	31,80	33,51	36,09	37,34	38,13	39,10	39,68	40,35	40,35	41,14	41,14
4.2.02	*CALIFORNIA*	512X512	26,47	27,28	28,89	30,00	31,14	31,70	32,45	33,22	34,33	36,99	38,13	38,59	39,10
4.2.03	*CALIFORNIA*	512X512	19,93	20,34	21,15	21,96	22,44	22,84	23,26	23,82	24,63	25,44	26,17	26,58	27,06
4.2.04	*CALIFORNIA*	512X512	25,99	27,46	30,42	31,80	33,22	34,51	35,58	36,37	36,99	37,72	38,13	38,59	39,10
4.2.05	*CALIFORNIA*	512X512	21,61	23,40	26,99	28,99	29,87	31,32	32,33	33,66	34,33	34,91	36,99	38,59	39,10
4.2.06	*CALIFORNIA*	512X512	21,39	22,81	25,65	26,83	28,09	28,79	29,62	30,21	30,73	31,60	32,00	32,69	33,22
4.2.07	*CALIFORNIA*	512X512	24,65	27,82	28,79	30,42	31,23	31,90	33,22	33,98	35,12	35,83	36,37	36,99	37,34
5.1.09	*CALIFORNIA*	256X256	24,86	25,70	27,38	27,84	28,17	28,45	28,79	29,21	29,56	29,94	30,14	30,42	30,73
5.1.10	*CALIFORNIA*	256X256	17,95	18,64	20,00	21,36	22,26	23,08	23,77	24,35	24,95	25,28	25,75	26,04	26,28
5.1.11	*CALIFORNIA*	256X256	23,91	25,80	26,61	27,56	28,17	28,79	29,32	29,81	30,00	30,28	30,50	30,73	31,14
5.1.12	*CALIFORNIA*	256X256	21,58	22,65	24,79	25,96	26,34	26,67	27,84	28,74	28,99	29,38	29,68	29,87	30,28
5.1.13	*CALIFORNIA*	256X256	14,56	15,75	18,14	19,65	22,50	25,21	27,13	28,89	31,80	33,82	34,49	35,15	35,82
5.1.14	*CALIFORNIA*	256X256	20,67	21,61	23,49	24,49	25,37	26,12	26,73	27,09	27,72	28,31	28,69	29,05	29,44
5.2.08	*CALIFORNIA*	512X512	21,22	22,26	24,35	25,23	25,65	26,17	26,64	28,45	29,27	29,94	30,28	30,65	30,97
5.2.09	*CALIFORNIA*	512X512	19,17	20,26	22,44	23,03	23,93	24,61	25,08	25,53	26,20	26,73	27,27	27,64	28,04
5.2.10	*CALIFORNIA*	512X512	20,25	20,91	22,58	22,83	23,57	24,19	24,77	25,01	25,44	25,85	26,31	26,70	27,02

			PSNR LHE												
NOMBRE	**BIBLIOTECA**	**TAMAÑO**	**0,05**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**	**1,1**	**1,2**
5.3.01	*CALIFORNIA*	1024X1024	23,53	24,56	26,61	27,27	28,22	28,49	28,84	29,44	29,87	30,35	30,57	30,97	31,14
5.3.02	*CALIFORNIA*	1024X1024	21,87	22,74	24,49	25,51	26,28	26,70	26,93	27,27	27,80	28,13	28,49	28,84	29,27
7.1.01	*CALIFORNIA*	512X512	25,34	25,34	27,56	28,09	28,69	29,15	29,56	29,94	30,14	30,35	30,57	30,97	31,14
7.1.02	*CALIFORNIA*	512X512	26,73	28,74	29,74	30,00	30,21	30,65	30,89	31,14	31,23	31,60	32,00	32,22	32,45
7.1.03	*CALIFORNIA*	512X512	26,12	26,75	28,00	28,59	28,89	29,15	29,50	29,94	30,14	30,50	30,57	31,06	31,23
7.1.04	*CALIFORNIA*	512X512	26,28	27,04	28,54	29,10	29,32	29,87	30,42	30,73	30,97	31,06	31,50	31,60	31,90
7.1.05	*CALIFORNIA*	512X512	23,33	24,07	25,55	26,09	26,64	26,96	27,52	27,96	28,31	28,59	28,89	28,99	29,38
7.1.06	*CALIFORNIA*	512X512	23,46	24,20	25,68	26,17	26,73	27,06	27,68	28,04	28,35	28,64	28,94	29,05	29,38
7.1.07	*CALIFORNIA*	512X512	24,46	25,21	26,70	27,13	27,49	27,88	28,31	28,74	29,10	29,44	29,56	29,94	30,21
7.1.08	*CALIFORNIA*	512X512	27,06	28,40	28,89	29,32	29,62	29,87	30,14	30,42	30,57	30,73	30,97	31,23	31,50
7.1.09	*CALIFORNIA*	512X512	24,59	25,24	26,55	27,09	27,49	28,04	28,31	28,89	29,21	29,56	29,81	30,07	30,21
7.1.10	*CALIFORNIA*	512X512	26,28	26,28	28,59	28,99	29,56	29,81	30,35	30,65	30,81	31,14	31,41	31,60	31,80
7.2.01	*CALIFORNIA*	1024X1024	28,84	29,50	29,74	29,94	30,14	30,21	30,57	30,73	30,89	31,06	31,32	31,41	31,70
Boat	*CALIFORNIA*	512X512	22,84	23,51	25,75	26,04	26,70	27,13	27,49	28,64	29,50	29,87	30,21	30,57	30,89
Elaine	*CALIFORNIA*	512X512	25,41	26,42	28,45	28,84	29,15	29,38	29,56	29,74	29,87	30,14	30,28	30,57	30,73
Gray21	*CALIFORNIA*	512X512	29,05	30,42	30,57	31,14	31,50	31,60	31,80	32,00	32,11	32,33	32,57	32,69	33,08
House	*CALIFORNIA*	512X512	22,70	22,70	25,37	26,46	27,27	28,17	29,56	30,28	31,06	31,60	32,33	32,82	33,36
Numbers	*CALIFORNIA*	256X256	15,97	16,37	17,19	17,48	17,83	18,14	18,50	19,06	19,94	21,18	22,36	23,32	24,08
Ruler	*CALIFORNIA*	512X512	10,36	10,36	10,75	11,49	11,49	14,55	18,36	50,00	50,00	50,00	50,00	50,00	50,00
Testpat	*CALIFORNIA*	1024X1024	16,77	18,88	21,67	26,58	29,62	31,50	32,33	32,69	33,51	33,98	34,51	34,91	36,67

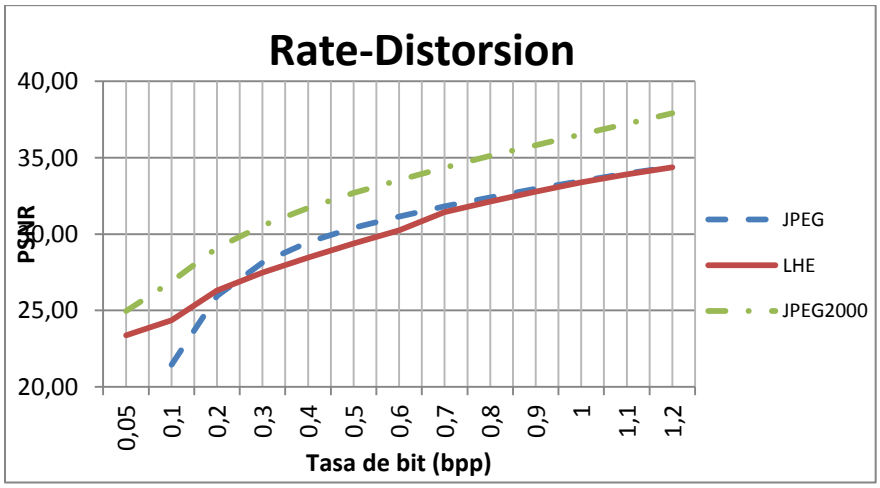
			PSNR JPEG											
NOMBRE	**BIBLIOTECA**	**TAMAÑO**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**	**1,1**	**1,2**
kodim01	*KODAK*	768x512	19,8	22,4	24,3	25,3	26,4	26,9	27,7	28,2	28,9	29,4	29,9	30,4
kodim02	*KODAK*	768x512	25,4	30,8	32,3	33,3	34,1	34,8	35,4	36	36,6	37,2	37,5	38,1
kodim03	*KODAK*	768x512	22,7	30,6	33,1	34,6	35,7	36,7	37,7	38,5	39,2	40	40,7	41,3
kodim04	*KODAK*	512x768	22,7	29,5	31,4	32,6	33,6	34,4	35,2	35,8	36,4	36,9	37,3	37,8
kodim05	*KODAK*	512x768	19,3	20,7	23,2	24,2	25,3	26,3	27	27,6	28,3	29	29,6	30,2
kodim06	*KODAK*	768x512	20,9	24,9	26,2	27,6	28,6	29,3	30,1	30,9	31,4	32,2	32,8	33,4
kodim07	*KODAK*	768x512	22	27,6	30,5	32,5	33,8	34,9	36	36,9	37,7	38,4	39,2	39,9
kodim08	*KODAK*	768x512	18,7	20,2	22	23,6	24,6	25,5	26,4	27,1	27,7	28,4	29	29,6
kodim09	*KODAK*	512x768	22,2	29,2	32	33,8	35,1	36,1	36,9	37,5	38,3	38,8	39,2	39,6
kodim10	*KODAK*	512x768	22,1	28,8	31,2	32,9	34,1	35,2	36	36,8	37,5	38,2	38,7	39,1
kodim11	*KODAK*	768x512	21,6	25,8	27,7	29,1	30	30,7	31,5	32,2	32,9	33,5	34,2	34,7
kodim12	*KODAK*	768x512	22,8	30,4	32,5	33,8	34,9	35,8	36,6	37,3	37,9	38,5	39	39,6
kodim13	*KODAK*	768x512	18,7	20,7	21,8	22,6	23,5	24,1	24,6	25,1	25,6	26,1	26,5	27
kodim14	*KODAK*	768x512	20,8	24,4	26,4	27,6	28,5	29,3	29,9	30,6	31,1	31,7	32,2	32,7
kodim15	*KODAK*	768x512	22,9	29,3	31,4	32,8	33,8	34,7	35,5	36,3	36,9	37,7	38,2	38,6
kodim16	*KODAK*	768x512	22,3	28,6	30,2	31,3	32,3	33,2	34	34,6	35,4	36	36,6	37,2
kodim17	*KODAK*	512x768	22,3	27,9	30	31,5	32,6	33,7	34,4	35,2	35,9	36,6	37,3	37,7
kodim18	*KODAK*	512x768	20,7	24	25,8	27	27,8	28,6	29,4	30	30,7	31,2	31,9	32,4
kodim19	*KODAK*	512x768	21,5	26,1	28,4	29,6	30,7	31,6	32,4	33,1	33,8	34,4	35	35,5
kodim20	*KODAK*	768x512	22	29,2	31,3	32,9	34,1	35,2	36,2	37	37,7	38,5	39,3	39,9
kodim21	*KODAK*	768x512	21,3	25,4	27,1	28,7	29,8	30,7	31,5	32,2	32,9	33,5	34,2	34,6
kodim22	*KODAK*	768x512	21,7	27	28,8	30,1	31,1	31,8	32,5	33,2	33,7	34,3	34,8	35,4
kodim23	*KODAK*	768x512	23,1	32,1	35	36,7	38	39,1	39,8	40,5	41,2	41,6	42,1	42,6
kodim24	*KODAK*	768x512	20,3	23,6	25,5	26,4	27,5	28,3	29,2	29,8	30,6	31,2	31,9	32,5

			PSNR JPEG											
NOMBRE	**BIBLIOTECA**	**TAMAÑO**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**	**1,1**	**1,2**
4.1.01	*CALIFORNIA*	256X256	22,8	27,2	30,6	32,7	34	35	35,7	36,4	37,1	37,6	38,1	38,5
4.1.02	*CALIFORNIA*	256X256	22,4	27,3	30,7	32,6	33,9	34,9	35,8	36,6	37,4	38	38,7	39,1
4.1.03	*CALIFORNIA*	256X256	23,2	31,1	35,1	37,6	39	40,1	40,9	41,4	42	42,4	43	43,4
4.1.04	*CALIFORNIA*	256X256	22,2	26,3	30,8	32,9	34,5	35,7	36,6	37,5	38,3	38,8	39,3	39,9
4.1.05	*CALIFORNIA*	256X256	22,8	25,9	30,5	32,7	34,1	34,9	35,6	36,4	37,1	37,9	38,5	39
4.1.06	*CALIFORNIA*	256X256	19,4	19,4	23,7	25,5	26,9	27,8	28,7	29,5	30	30,6	31	31,5
4.1.07	*CALIFORNIA*	256X256	22,8	30,2	35,6	38,8	40,7	42,2	43,4	44,3	45,2	46	46,4	46,8
4.1.08	*CALIFORNIA*	256X256	22,3	26,1	31,6	34,6	36,8	38,4	39,5	40,4	41,4	42,3	43,2	43,8
4.2.01	*CALIFORNIA*	512X512	23	32,7	35,5	36,9	37,9	38,7	39,4	39,9	40,4	40,8	41,1	41,6
4.2.02	*CALIFORNIA*	512X512	23,7	30	32,4	33,6	34,5	35,2	35,9	36,4	36,9	37,4	37,9	38,3
4.2.03	*CALIFORNIA*	512X512	19	20,9	22	22,8	23,7	24,3	25	25,4	25,9	26,4	26,8	27,3
4.2.04	*CALIFORNIA*	512X512	21,9	28,9	31,7	33,4	34,6	35,4	36,2	36,7	37,3	37,8	38,2	38,5
4.2.05	*CALIFORNIA*	512X512	21	27,5	30,5	32,4	33,9	35,2	36	36,8	37,6	38,2	38,8	39,3
4.2.06	*CALIFORNIA*	512X512	20,9	24,3	27,5	29	29,9	30,8	31,5	32,1	32,5	32,9	33,3	33,6
4.2.07	*CALIFORNIA*	512X512	22,2	28	31,3	32,8	33,8	34,5	35	35,4	35,8	36,1	36,4	36,7
5.1.09	*CALIFORNIA*	256X256	22,2	26,2	27,6	28,2	28,6	28,9	29,1	29,4	29,6	29,9	30,1	30,3
5.1.10	*CALIFORNIA*	256X256	18,4	18,4	21	22,4	23,3	24	24,8	25,3	26	26,3	26,8	27,2
5.1.11	*CALIFORNIA*	256X256	22	27,2	29,7	30,6	31	31,2	31,4	31,6	31,8	31,9	32,1	32,3
5.1.12	*CALIFORNIA*	256X256	21,5	23,1	26,5	28,1	28,9	29,4	29,7	30	30,2	30,4	30,7	30,9
5.1.13	*CALIFORNIA*	256X256	15,8	19,5	22,1	24,2	25,7	27,5	29,2	30,7	31,9	33,2	34,3	35,5
5.1.14	*CALIFORNIA*	256X256	19,9	21,5	24,1	25,4	26,2	27	27,6	28,1	28,5	28,8	29,2	29,5
5.2.08	*CALIFORNIA*	512X512	21,1	24,4	26,6	27,7	28,5	29,1	29,6	30,1	30,4	30,8	31	31,2
5.2.09	*CALIFORNIA*	512X512	19,2	20,8	23,3	24,5	25,4	26	26,7	27,3	27,7	28,1	28,5	28,8
5.2.10	*CALIFORNIA*	512X512	19,7	21,8	23,3	24	24,7	25,3	25,7	26,1	26,6	26,9	27,2	27,5

			PSNR JPEG											
NOMBRE	**BIBLIOTECA**	**TAMAÑO**	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**	**1,1**	**1,2**
5.3.01	*CALIFORNIA*	1024X1024	21,4	25,9	27,8	28,6	29,3	29,7	30	30,3	30,6	30,8	31	31,3
5.3.02	*CALIFORNIA*	1024X1024	20,5	24,7	25,9	26,8	27,3	27,7	28	28,3	28,6	28,9	29,1	29,3
7.1.01	*CALIFORNIA*	512X512	22,8	27,1	28,2	29	29,4	29,8	30,1	30,4	30,6	30,8	31	31,2
7.1.02	*CALIFORNIA*	512X512	25,4	30	30,6	30,8	31	31,1	31,3	31,4	31,6	31,8	32	32,2
7.1.03	*CALIFORNIA*	512X512	21,9	27,4	28,4	28,9	29,3	29,6	29,9	30,2	30,4	30,6	30,8	31
7.1.04	*CALIFORNIA*	512X512	22,6	27,5	28,8	29,4	29,9	30,4	30,7	30,9	31,2	31,4	31,6	31,7
7.1.05	*CALIFORNIA*	512X512	21,1	24,8	25,8	26,7	27,3	27,8	28,1	28,5	28,8	29,1	29,3	29,5
7.1.06	*CALIFORNIA*	512X512	21,2	24,8	25,9	26,8	27,3	27,8	28,2	28,5	28,8	29,1	29,4	29,6
7.1.07	*CALIFORNIA*	512X512	21,5	25,7	26,8	27,4	27,9	28,3	28,7	28,9	29,2	29,5	29,7	29,9
7.1.08	*CALIFORNIA*	512X512	24,3	28,8	29,4	29,8	30	30,3	30,5	30,7	30,9	31	31,2	31,4
7.1.09	*CALIFORNIA*	512X512	21,4	25,8	26,9	27,8	28,2	28,6	29,1	29,4	29,6	29,9	30,2	30,5
7.1.10	*CALIFORNIA*	512X512	22,8	27,4	28,6	29,4	29,9	30,2	30,5	30,8	31	31,2	31,4	31,6
7.2.01	*CALIFORNIA*	1024X1024	27	29,7	30	30,2	30,4	30,6	30,8	31	31,2	31,4	31,5	31,8
Boat	*CALIFORNIA*	512X512	20,8	24,6	26,7	27,7	28,5	29,1	29,5	29,9	30,2	30,4	30,7	30,9
Elaine	*CALIFORNIA*	512X512	22,4	27,5	28,7	29,1	29,4	29,6	29,8	29,9	30,1	30,3	30,4	30,5
Gray21	*CALIFORNIA*	512X512	24,1	31,5	31,7	32	32,2	32,5	32,6	32,8	33,1	33,5	33,7	34,1
House	*CALIFORNIA*	512X512	20,6	25,2	27,6	29,3	30,7	31,8	32,8	33,6	34,3	35	35,7	36,4
Numbers	*CALIFORNIA*	256X256	16,6	17,6	18,7	19,6	20,6	21,3	22,4	23,1	23,9	24,6	25,3	25,9
Ruler	*CALIFORNIA*	512X512	12,5	16,2	16,2	18,9	19,8	20,9	21,7	22,7	24	25,2	27,1	27,6
Testpat	*CALIFORNIA*	1024X1024	19,9	24,2	28,5	31,2	32,6	32,7	33,9	36,4	37,4	38,2	40,9	42,2

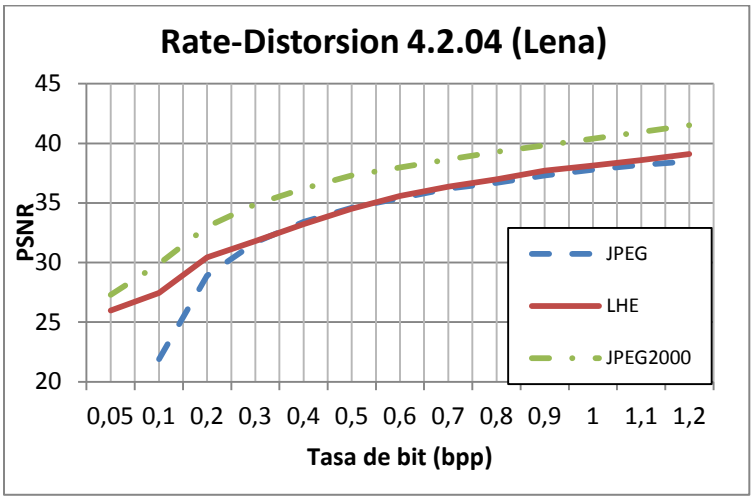
Para realizar la comparativa entre LHE y JPEG, se ha procedido a realizar un estudio del PSNR medio. Para ello, se realiza el PSNR promedio para cada tasa de bit. Esta medida es una forma experimental de comparar dos tipos de codificación distinta para poder observar los resultados que ofrece nuestro algoritmo. En la tabla se puede observar el resultado numérico mientras que en la gráfica podemos ver, exactamente, los puntos en los que ofrecemos mejores resultados y en los que aún hay que trabajar para mejorarlos.

	BIT-RATE MEDIO												
	0,05	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**	**1,1**	**1,2**
JPEG		21,44	25,95	28,13	29,46	30,40	31,15	31,82	32,41	32,96	33,46	33,97	34,40
LHE	23,37	24,35	26,30	27,48	28,46	29,38	30,25	31,43	32,14	32,79	33,39	33,90	34,38



Como se puede observar, LHE es mucho mejor que JPEG comprimiendo a muy bajos bit por píxel. Además, a partir de 1 bpp ofrece resultados muy similares. Sin embargo, existe una caída muy pronunciada de la curva LHE desde 0,7 bpp hasta 0,3 bpp. Este problema se debe a que las métricas de Relevancia Perceptual y los escalados no se aplican de forma eficiente en estos puntos de forma que la calidad cae con respecto a JPEG. Este es un punto a mejorar del algoritmo en un trabajo futuro.

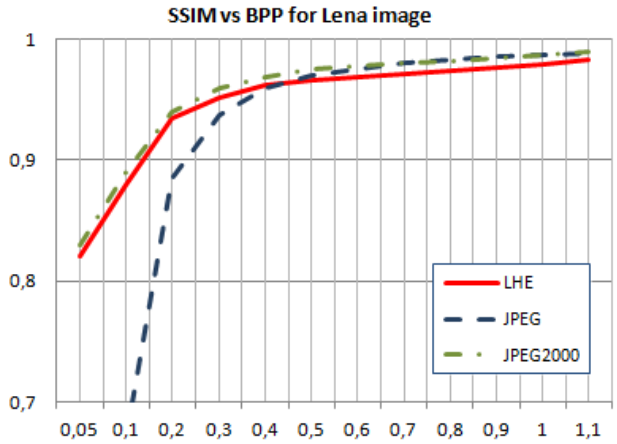
A continuación, se verá el ejemplo de una imagen que ha sido muy representativa en los estudios de imagen, Lena. Se trata de la imagen 4.2.04 de la biblioteca obtenida de USC-SIPI. Con los resultados de las tablas anteriores, su gráfica Rate-Distorsion es la que se muestra a continuación:



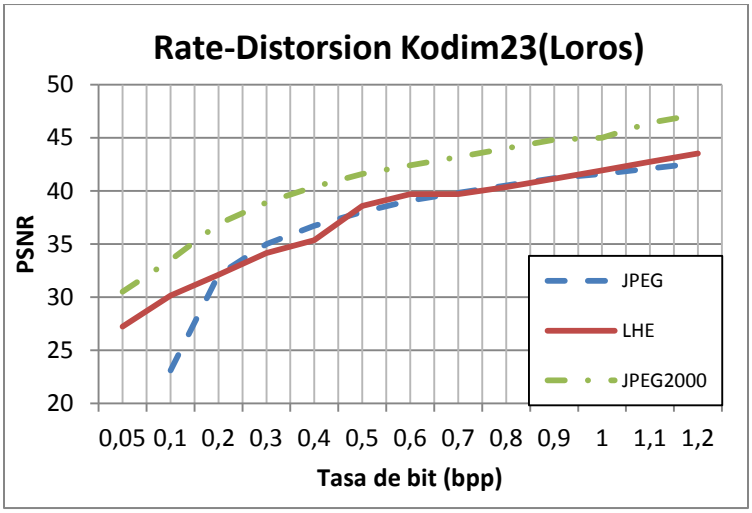
Como se puede observar, los resultados de la imagen de Lena son realmente buenos ya que son muy similares a los de JPEG e, incluso, lo supera en muchos puntos. A continuación, se muestran los resultados en imágenes de codificar con LHE, utilizando diferentes tasas de bit, la imagen de Lena.

1.2 bpp **0.5 bpp** **0.1bpp**

Además, se incluye la medida de SSIM para la imagen de Lena. Tal y como se puede observar, los resultados son realmente buenos ya que, para dicha imagen, son muy similares a los que se derivan de la utilización de JPEG2000.



Otra imagen que suele aparecer en los estudios de imagen es la que en este documento se llama Kodim23, de la biblioteca de imágenes de Kodak, también conocida como Loros. Los resultados de las tablas anteriores permiten esbozar la siguiente gráfica:

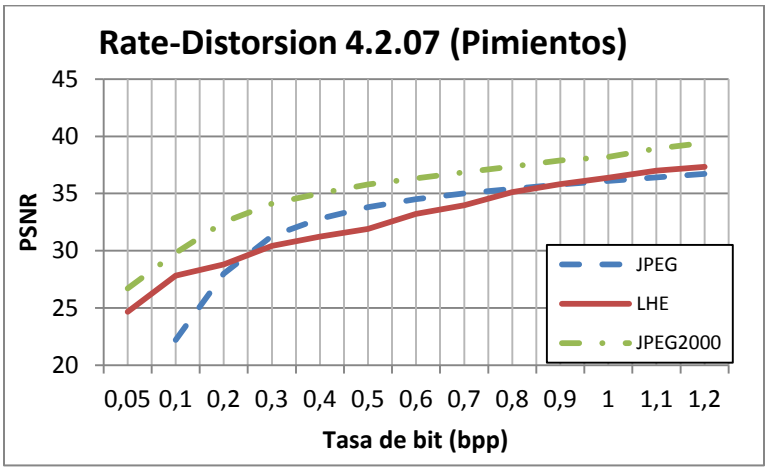


Al igual que en Lena, la imagen de los Loros, comprimida con LHE, presenta una caída de calidad con respecto a JPEG en los puntos de 0,2 a 0,5 bpp. La explicación es la misma que la anterior. No obstante,

para el resto de tasas de bit, se logra un resultado mejor o similar al que se consigue con JPEG. A continuación, se muestran los resultados en imágenes de codificar con LHE, utilizando diferentes tasas de bit, la imagen de los Loros.

1.2 bpp **0.5 bpp** **0.1bpp**

Por último, se muestra otra imagen contenida en la librería y que también ha sido muy utilizada en los estudios de imagen. Se trata de la imagen 4.2.07, más conocida como Pimientos. Utilizando los resultados de las tablas anteriores, se dibuja la gráfica Rate-Distorsion que aparece a continuación:



Tal y como ocurre en las imágenes anteriores, la imagen de los Pimientos, comprimida con LHE, presenta una caída de calidad con respecto a JPEG en los puntos de 0,3 a 0,8 bpp. La explicación es la misma que la anterior. No obstante, para el resto de tasas de bit, se logra un resultado mejor o similar al que se consigue con JPEG. A continuación, se muestran los resultados en imágenes de codificar con LHE, utilizando diferentes tasas de bit, la imagen de los Pimientos.

1.2 bpp **0.5 bpp** **0.1bpp**

En cuanto a resultados en la compresión, una característica destacable que diferencia a LHE de JPEG es que el primero permite comprimir a bajos bit-rates con una calidad muchísimo mejor que la que se consigue con JPEG. Esta característica se puede observar en las gráficas anteriores pero, además, a continuación, se incluyen las imágenes de los ejemplos anteriores codificadas a 0.1 bpp con JPEG y LHE para que el lector pueda comparar los resultados visualmente.

Lena JPEG **Loros JPEG** **Pimientos JPEG**

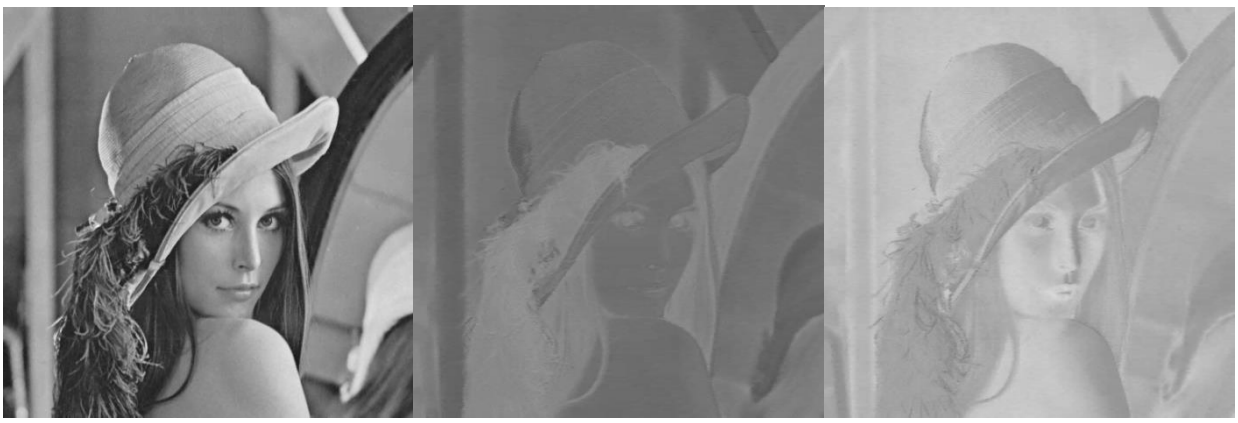
Lena LHE **Loros LHE** **Pimientos LHE**

5.1 Resultados imágenes en color

A continuación, se muestran ejemplos de imágenes en color. En primer lugar, se muestran los resultados de comprimir en color las tres imágenes que se han tomado de ejemplo en los apartados anteriores utilizando el modelo YUV 4:2:2 para LHE. Así, se muestran, mediante imágenes, los escalados que se producen en cada una de las componentes de color, dichas componentes de color reescaladas y la imagen resultado final, fruto de la mezcla de las tres componentes. Para el modelo YUV 4:2:0 se procede de la misma manera. Se puede observar en las imágenes que los escalados que tienen lugar cuando se utiliza este último modelo son mayores. Esto se debe a que cada una de las señales de crominancia se escalan a la mitad que la señal de luminancia, tanto verticalmente como horizontalmente. En el modelo YUV 4:2:2, sin embargo, solamente se escalan a la mitad horizontalmente con respecto a la señal de luminancia. Como consecuencia, las imágenes en color del modelo YUV 4:2:0 ofrecen un ratio de compresión mayor pero también disminuyen su calidad frente a las que se obtienen utilizando YUV 4:2:2.

5.1.1 YUV 4:2:2

- Lena



Luminancia Y **Crominancia U** **Crominancia V**

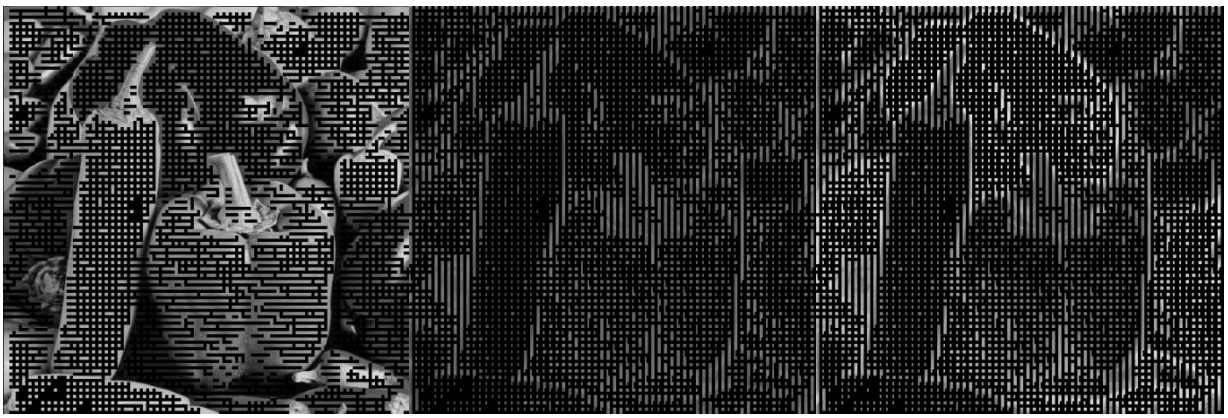
Lena a color con una tasa de bit de 2 bpp y un PSNR de 33,66 dB

- Loros

Luminancia Y **Crominancia U** **Crominancia V**

Loros a color con una tasa de bit de 2 bpp y un PSNR de 36,09 dB

- Pimientos

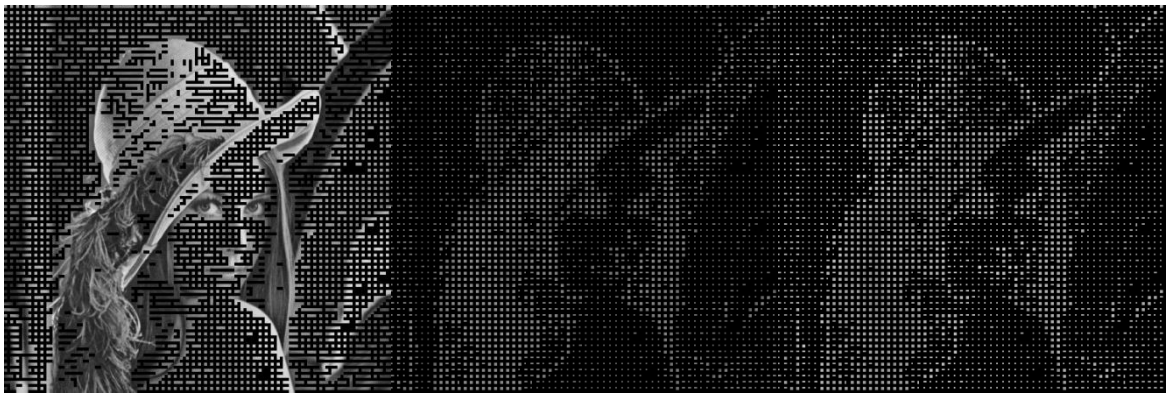


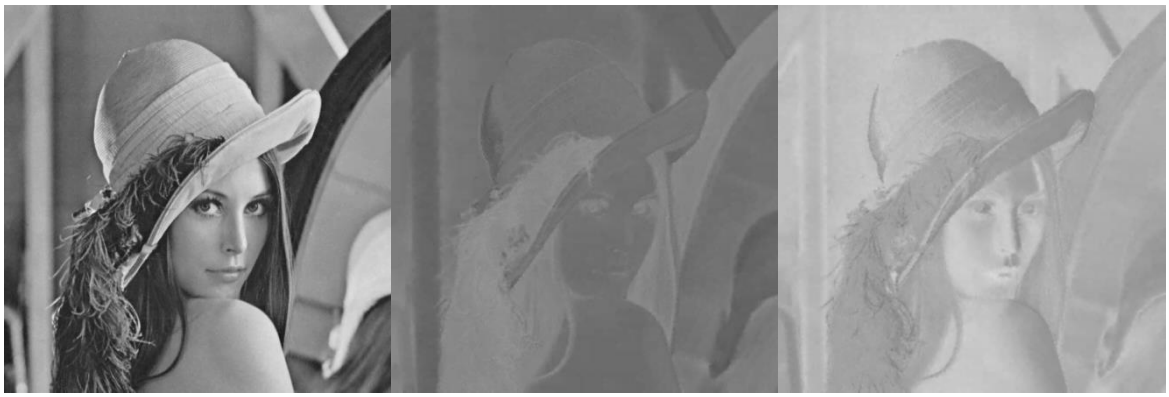
Luminancia Y **Crominancia U** **Crominancia V**

Pimientos a color con una tasa de bit de 2 bpp y un PSNR de 31,41 dB

5.1.2 YUV 4:2:0

- Lena

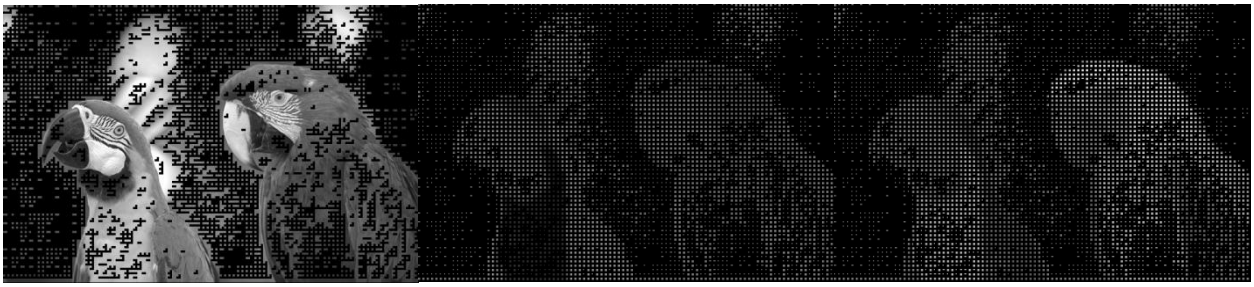




Luminancia Y **Crominancia U** **Crominancia V**

Lena a color con una tasa de bit de 1,6 bpp y un PSNR de 33,22 dB

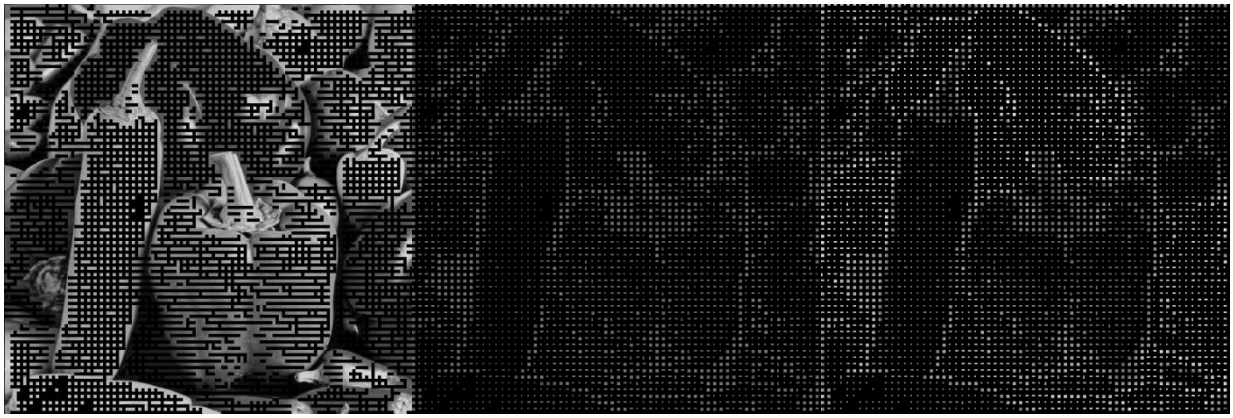
- Loros

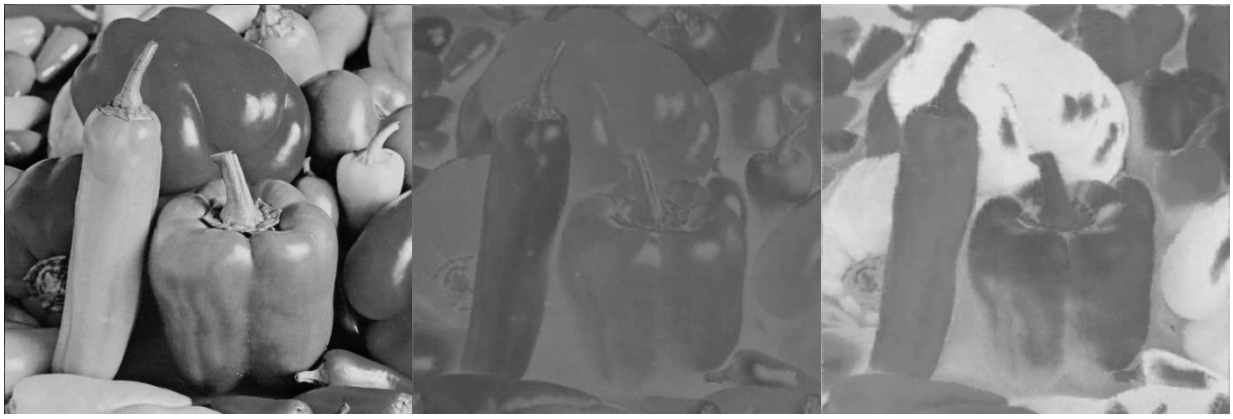


Luminancia Y **Crominancia U** **Crominancia V**

Loros a color con una tasa de bit de 1,37 bpp y un PSNR de 35,82 dB

- Pimientos





Luminancia Y **Crominancia U** **Crominancia V**

Pimientos a color con una tasa de bit de 1,55 bpp y un PSNR de 30,14 dB

6 Video basado en LHE

En el apartado de Estado del Arte 2.4 Codificación de vídeo, se vio que $\frac{2}{3}$ del tiempo total de codificación en los sistemas de vídeo actuales se pierde en la estimación de vectores de movimiento. Esto es debido a que, normalmente, se realiza una operación de comparación entre un bloque de un fotograma y bloques situados alrededor en los fotogramas anteriores y/o posteriores. Es decir, una convolución matemática, muy costosa. De esta manera, muchas de las investigaciones actuales dentro el campo de la codificación de vídeo se centran en buscar algoritmos que puedan realizar más eficientemente la estimación de movimiento. El video basado en LHE nace, precisamente, con este propósito.

6.1 Hop logarítmico temporal

En apartados anteriores, se ha visto que un *hop*, en la imagen fija, se define como un salto logarítmico respecto de la predicción de luminancia que se realiza para cada píxel, de forma que dicho *hop* permite ajustar el error cometido en la predicción. En el caso del vídeo, el concepto es similar pero la relación se establece entre *frames*. Es decir, un *hop* en vídeo es un salto logarítmico con respecto a la luminancia del mismo píxel del fotograma anterior. Llamaremos a los *hops* temporales *T-hops*.

Para comprimir la imagen fija, se utilizan métricas de Relevancia Perceptual con el objetivo de averiguar la relevancia de información que existe en cada bloque y, en función de los resultados obtenidos, realizer el *downsampling* de la imagen. En el caso del vídeo, sin embargo, las métricas que se aplican sobre los *T-hops* indican cuánto ha cambiado la información entre los *frames* de vídeo, es decir, cómo se ha modificado la imagen de un fotograma a otro. De esta manera, un *hop* nulo indica que la luminancia es la misma que en el *frame* anterior, mientras que un *hop* grande significa que la luminancia se ha modificado.

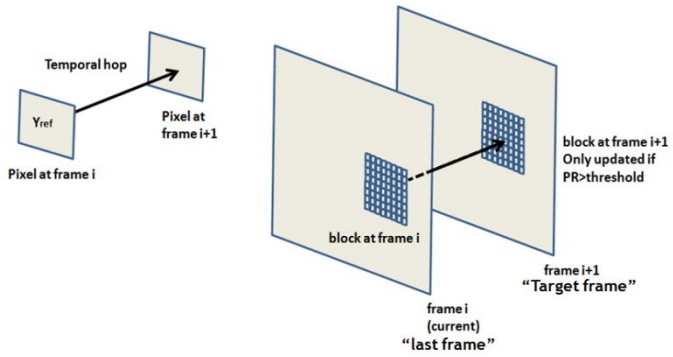


Figura 46. Hop logarítmico temporal

Así, se puede representar una secuencia de video con *T-hops*. Sin embargo, si existe un movimiento muy brusco, se pierden los detalles ya que incluso los *hops* más grandes no son capaces de alcanzar la nueva luminancia del píxel en el nuevo *frame*. No obstante, cuando la imagen se estabiliza, se comienzan a distinguir los detalles puesto que los nuevos *hops*, más pequeños, van refinando en el tiempo la imagen.

En la Figura 46 se aprecia el efecto de cuantización que tiene lugar cuando existe un movimiento rápido en un *frame*. En este caso, se trata de la mano de Foreman, que aparece rápidamente en la pantalla. Como se puede observar, mediante un solo *hop*, no se alcanza con precisión el valor de la luminancia deseada. Este efecto negativo deberá ser estudiado en el futuro, de cara a paliarlo o eliminarlo en la medida de lo posible. Por otro lado, cuando los movimientos no son bruscos, el efecto de cuantización no resulta evidente para el observador.

Figura 47. Efecto de cuantización

A continuación, en la Figura 47, se muestra una imagen con un motivo floral codificada con *hops* temporales. Si la imagen permanece fija, sin movimiento, inicialmente aparecen áreas conexas del mismo tipo de *hop*, denominadas nubes de *hops*, y a medida que se envían más fotogramas los *hops* son cada vez más pequeños llegando a enviarse solo *hops* nulos al cabo de unos tres fotogramas. Este efecto se puede observar en el siguiente ejemplo, en el que los detalles de la imagen se van refinando y, al cabo de tres fotogramas, prácticamente solo existen *hops* nulos ya que se ha alcanzado el valor real de luminancia de cada píxel del *frame*. De esta manera, en el primer fotograma aparecen los *hops* del conjunto

$$H(x) = \{h_{-4}, h_{-3}, h_{-2}, h_{-1}, h_0, h_1, h_2, h_3, h_4\} \quad (22)$$

En la Figura 47, estos *hops* aparecen representados por el subíndice que les corresponde. El *hop* nulo h_0 también aparece en este fotograma porque la luminancia de algunos píxeles es igual a la del *frame* de referencia. El *frame* de referencia que se utiliza para este primer fotograma está formado por píxeles cuyo valor de luminancia es Y=128.

En el segundo fotograma utiliza como *frame* de referencia el fotograma anterior y los *hops* grandes h_{-4} y h_4 han desaparecido. Además, aparecen los *hops* no nulos h_{-2}, h_{-1}, h_1 y h_2 que aproximan la luminancia del fotograma anterior aún más a la luminancia real del fotograma actual.

Finalmente, en el tercer *frame* todos los *hops* son nulos puesto que la imagen se ha estabilizado. Es decir, al no haber habido movimientos muy bruscos, los *hops* han conseguido, finalmente, adaptar la luminancia del *frame* codificado a la luminancia original.

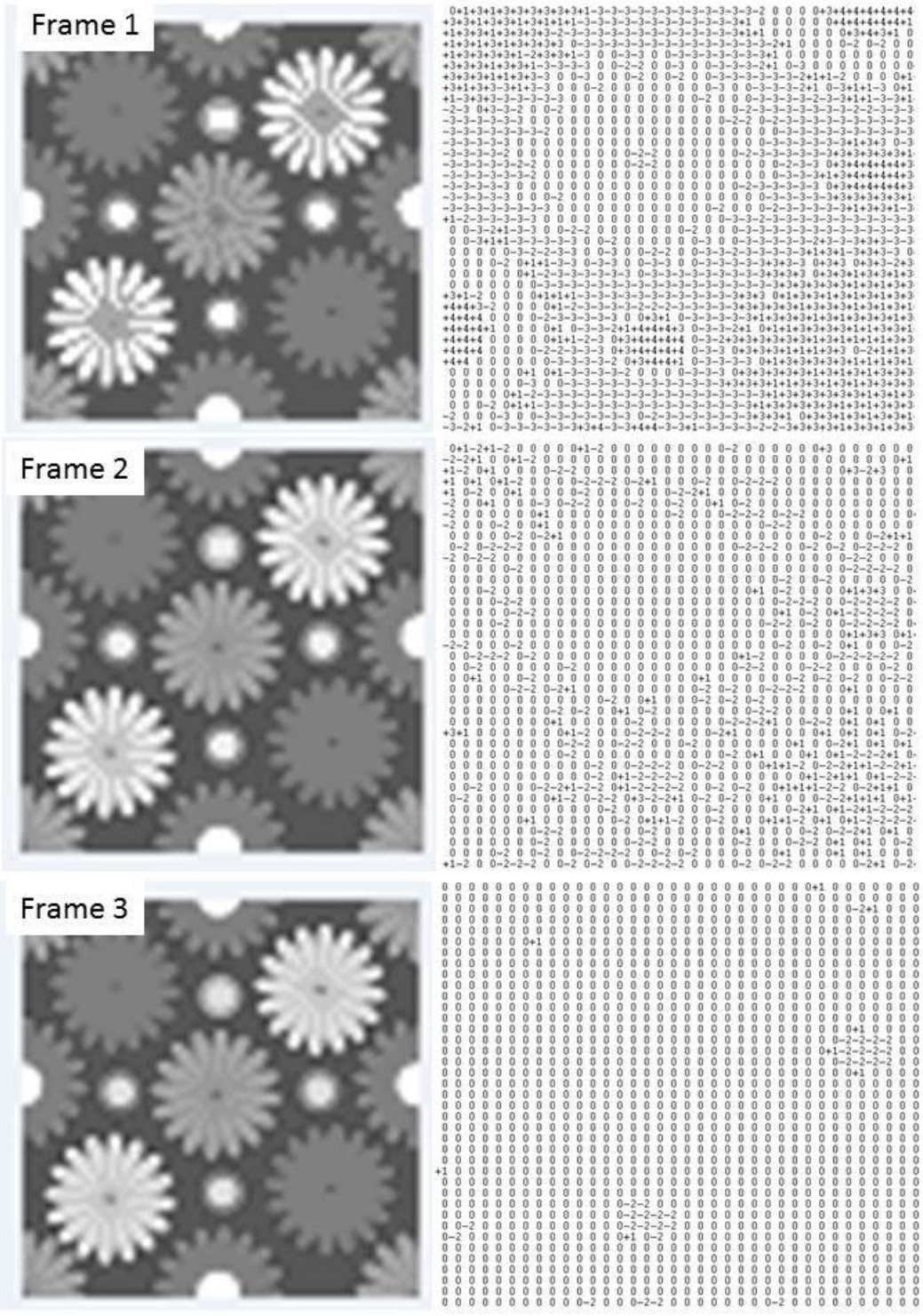


Figura 48. Codificación de vídeo.

El vídeo ejemplo de la Figura 48, *Foreman* ha sido calculado solo con hops temporales. De esta manera, se ha puesto a prueba el concepto de codificación de vídeo basado en estos *hops* y los resultados han sido satisfactorios tal y como se puede observar. Es decir, es posible codificar video con resultados aceptables usando T-hops.

Figura 49. Vídeo Foreman

En la película a la que corresponden estos fotogramas, la calidad aumenta cuando la imagen se estabiliza, mientras que los movimientos bruscos del personaje o los cambios rápidos de plano por parte de la cámara son peor codificados.

No se han realizado evaluaciones de calidad usando herramientas específicas de video pero, con este experimento, se ha puesto a prueba con éxito el concepto lo que sienta las bases para la creación de un codificador basado en este algoritmo.

6.2 Información Residual

Si se analizan los *hops* temporales de un objeto liso desplazándose por la pantalla, se puede observar cómo los píxeles correspondientes a los bordes del objeto se codifican con *hops* que permiten transformar el valor de la luminancia de fondo en el color del objeto. De esta forma, se pueden identificar claramente los límites del objeto en desplazamiento.

Así, en la Figura 49, se pueden diferenciar dos áreas de *hops* no nulos (representados mediante "." para facilitar su visualización) localizadas a la derecha e izquierda del círculo en movimiento. Estas áreas formadas por *hops* del mismo tipo se denominarán *nubes*. De esta manera, las *nubes* se irán desplazando conforme lo haga el círculo, siendo esta información la única relevante para codificar la evolución de un fotograma al siguiente. Es lo que se denominará I*nformación Residual*. Por tanto, la *Información Residual* está compuesta por los hops temporales que permiten que las luminancias del *frame* i se transformen en las del *frame* i+1.

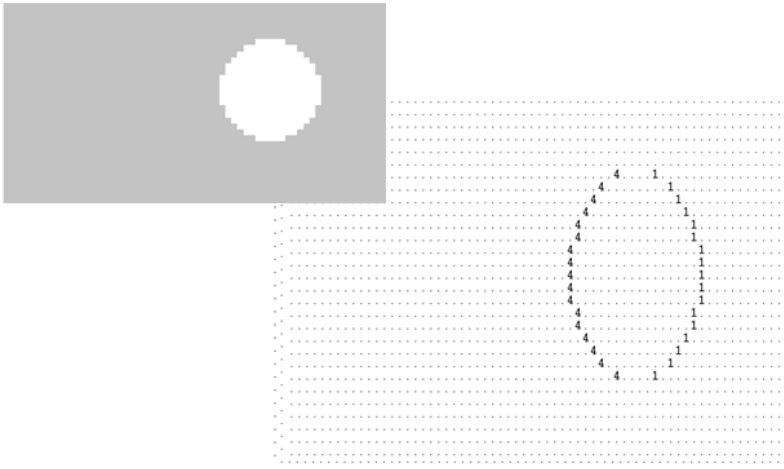


Figura 50. Información Residual

6.3 Nubes

El concepto de *nube* de hops ha sido introducido brevemente en el subapartado anterior. De esta manera, una *nube* se define como un conjunto de *hops* del mismo tipo conectados entre sí. Las conexiones son adyacencias entre hops.

6.3.1 Algoritmo de detección de nubes

Para poder localizar las *nubes* de cada fotograma, se ha ideado un algoritmo de detección de *nubes* que recorre únicamente una vez los hops de cada bloque por lo que se trata de un algoritmo lineal.

En la Figura 50 aparece un bloque con un motivo floral y sus T-hops asociados respecto de las luminancias del *frame* de referencia. En este *frame* de referencia, todos los píxeles toman el valor Y=128. Es decir, los T-hops que aparecen en este ejemplo son el resultado de aproximar, desde la luminancia Y=128, la luminancia que existe en el bloque que aparece en la imagen. Algunas de las *nubes* que aparecen han sido resaltadas.

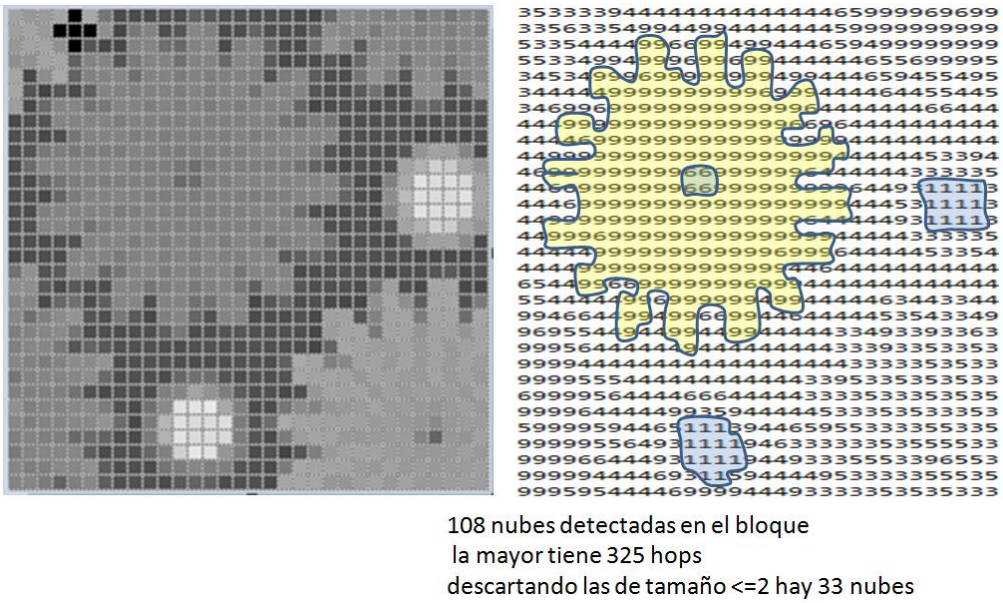


Figura 51. Fotograma y *nubes* de *hops*.

Debido a que habrá tantos tipos de *nubes* como tipos de *hops*, es recomendable agrupar en una misma *nube* los *hops* similares. Por ejemplo, son similares los *hops* pequeños h_{-1} y h_1 así como el *hop* nulo h_0. De esta forma, se agrupan tres tipos de *hops* en el mismo tipo de *nube* lo que permite reducir, en gran medida, el número de nubes detectadas en el bloque y simplificar los cálculos. El algoritmo de detección de nubes procede de la siguiente manera:

1. Codificación de T-hops de un bloque con respecto al *frame* de referencia.
2. Se recorren, uno a uno, los *hops*. Para ello, se recorre la imagen línea a línea.
3. Para cada hop:

a. Se comprueba si hay ya una *nube* creada en algún pixel adyacente que sea del mismo tipo de *hop*.
b. Si la hay, se introduce el *hop* en esa *nube*.
c. Si no la hay, se crea una nueva *nube* y se introduce el *hop*.
d. Con independencia de la *nube* donde se haya introducido el *hop*, se enlazan las *nubes* del mismo tipo de *hop* que sean adyacentes a ese *hop*.

Una vez recorrida toda la imagen, se obtiene una lista de *nubes* en la que muchas de ellas están enlazadas. Para hacer la lista definitiva, se crea una nueva en la que se añaden las *nubes* de la lista inicial así como todas sus enlazadas. Al fusionar una *nube* con otra, la *nube* principal absorbe todos los *hops* de las *nubes* con las que estaba enlazada mientras que estas últimas pierden todos sus *hops*. De esta manera, si se encuentra una *nube* que no contiene *hops* significa que ya se ha fusionado con otra, por lo que pasamos a la siguiente. El resultado de este proceso es una lista de *nubes* sin enlaces y donde cada una posee un solo tipo de *hop*.

El algoritmo anterior ha sido probado satisfactoriamente.

Por último, es necesario indicar que las *nubes* que contienen un número muy pequeño de *hops* (uno o dos), no sirven para calcular el movimiento ya que es posible que se trate de simple ruido y, por tanto, no es posible estimar el movimiento a partir de ellas. Además, no conviene considerar píxeles aislados para evaluar un movimiento que involucra a todo un bloque de la imagen. Por esta razón, se pueden descartar algunas *nubes* de la lista antes de seleccionar la mejor de ellas para estimar el vector de movimiento.

6.4 Vectores de movimiento e Información Residual

Identificar las *nubes* y relacionar cada *nube* con la *nube* correspondiente en el siguiente fotograma, permite calcular el vector de movimiento con un coste de cálculo mínimo.

Como se ha estudiado, no es possible usar la Información Residual para este propósito puesto que los *hops* necesarios para saltar del *frame* i al *frame* i+1 pueden ser muy diferentes de los necesarios para saltar del i+1 al i+2. Todo depende del contenido del *frame*. En la Figura 51 se muestra un ejemplo en el que un bloque se desplaza horizontalmente y la Información Residual que se produce con este movimiento.

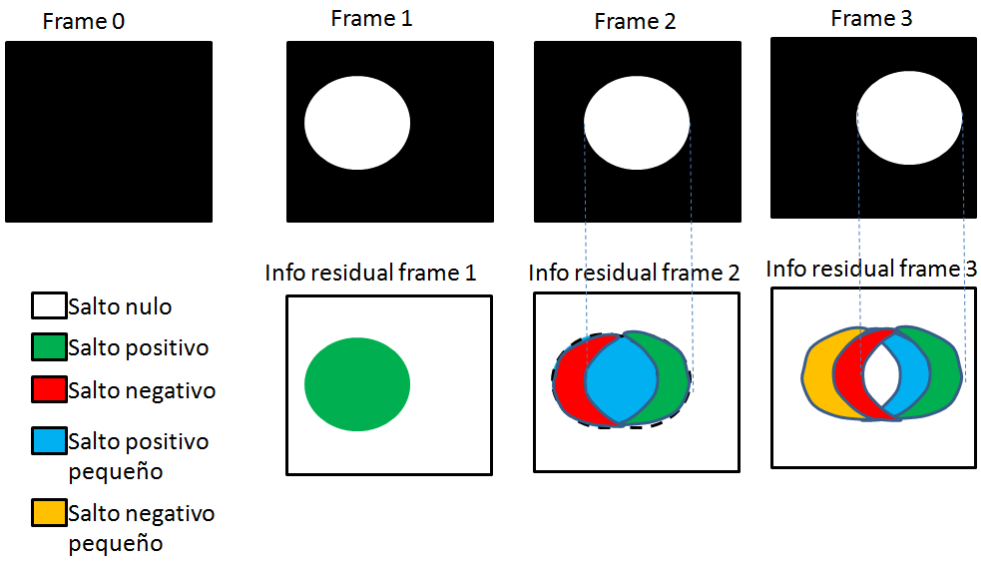


Figura 52. Información Residual

La pelota blanca hace su aparición en el *frame* 1. Lo píxeles del fondo coinciden con el color del frame 0, de modo que serán codificados con *hops* nulos h_0. Sin embargo, el área cubierta por la pelota será codificada con *hops* positivos que permiten incrementar la luminancia con respecto al *frame* anterior para alcanzar al color blanco.

En el *frame* 2, la pelota se ha desplazado. Ahora, en la franja negra que la pelota ha abandonado hay que retornar al color negro y, por lo tanto, hacen falta *hops* negativos en esa zona que permitan modificar la luminancia con respecto al *frame* anterior y, así, pasar del blanco al negro. Por el contrario, en la zona negra que la pelota está invadiendo, es necesario codificarla mediante *hops* positivos para poder pasar del negro al blanco. Por último, en el interior de la pelota hacen falta *hops* positivos pequeños para terminar de alcanzar el blanco de la pelota que no se alcanzó en la codificación anterior.

En el siguiente *frame*, los píxeles que retornaron al negro en la codificación anterior con *hops* negativos, ahora tratan de terminar de ajustarse a dicho tono utilizando *hops* más pequeños por lo que se crea una estela en la imagen. Además, como en el caso anterior, aparece una franja de píxeles en la que es necesario usar *hops* negativos para saltar hacia el negro detrás de la pelota y otra en la que es necesario codificar con *hops* positivos para alcanzar el color blanco en la parte derecha de la pelota. Además, algunos píxeles que en el *frame* anterior habían sido codificados para permitir el salto desde el negro al

blanco tratan de terminar de alcanzar este tono por lo que se codifican con un *hop* pequeño. En total, esta vez tenemos cuatro nubes de diferentes tipos de *hops*.

Como se puede observar, las *nubes* de *hops* son diferentes en número y en tipo de *hop* que contienen por lo que compararlas no es suficiente para estimar el movimiento del objeto fácilmente. A medida que la imagen se complica, el número y variedad de *nubes* aumenta y correlar las *nubes* de un *frame* con las del *frame* anterior se convierte en una tarea ardua y difícil ya que las nubes son diferentes unas de otras y se pierde la relación entre ellas. Por esta razón, no se puede estimar lo que se ha movido la *nube* de un frame con respecto a su análoga en el anterior y, por tanto, tampoco se puede estimar el movimiento del bloque. Anteriormente, se ha visto un ejemplo sencillo de una bola blanca sobre un fondo negro en la que aún se podría realizar la correlación entre nubes y estimar el movimiento. Sin embargo, si el fondo no fuese negro, las *nubes* de *hops* positivos se transformarían en diferentes *nubes* de *hops* positivos y de *hops* negativos, según las luminancias que existen en el fondo. De esta manera, se complica aún más la tarea de correlar las nubes de un *frame* con las del *frame* anterior.

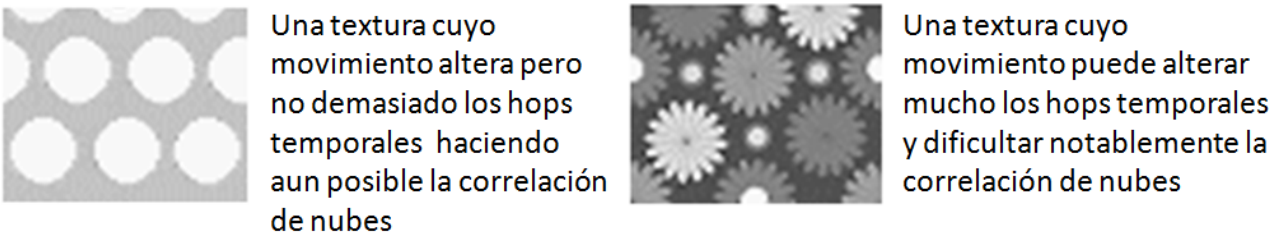


Figura 53. Ejemplo de teturas

Para solucionar este problema, es necesario modificar el *frame* de referencia. De esta forma, en lugar de utilizar el *frame* anterior, tal y como se ha venido haciendo hasta ahora, se utilizará como referencia un *frame* imaginario en el que todos los píxeles tienen el valor Y=128. El resultado de aplicar este *frame* se puede observar en la Figura 53.

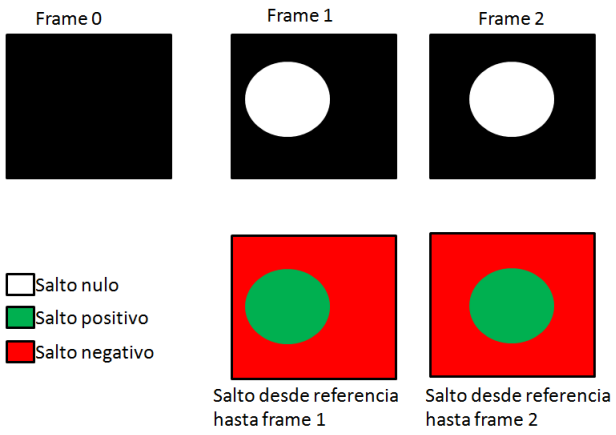


Figura 54. Hops utilizando *frame* de referencia Y=128

En esta ocasión, existen dos *nubes* de *hops* diferentes en cada frame: una correspondiente a la pelota y la otra correspondiente al fondo. La *nube* correspondiente a la pelota puede servirnos para estimar el movimiento. Para ello, simplemente debemos realizar la correlación de las *nubes*, es decir, encontrar su análoga en el *frame* siguiente y, de esta manera, calcular cómo se desplaza el centro geométrico de dicha *nube* de *hops*. Por nube análoga, se entiende una *nube* de tamaño y forma similar y del mismo tipo de *hop* que se encuentra en ambos *frames*.

Una vez calculado el vector de movimiento, se aplica dicho vector al frame i y, a continuación, se codifica la Información Residual entre el *frame* i+1 resultado de aplicar el vector de movimiento al *frame* i dando lugar a una aproximación del *frame* i+1y el *frame* i+1original.

Esta estrategia es mucho más fiable para calcular los vectores. Sin embargo, implica llevar a cabo dos codificaciones: la primera para calcular las nubes y estimar su movimiento y la segunda para codificar la Información Residual final, que enviaremos junto con el vector.

6.5 Estimación del vector de movimiento basado en *nubes*

La estimación de vectores de movimiento basada en *nubes* está aún en fase experimental. Se han probado algunas estrategias en las que se utiliza, para la estimación del movimiento, los cambios que se producen en las *nubes*. Esto permite evitar los costosos cálculos que suponen los mecanismos tradicionales de convolución en dos dimensiones que se utilizan para hallar los vectores de movimiento. De las estrategias

que se han probado, la que mejores resultados ha ofrecido es la basada en el cambio de posición del centro geométrico dela *nube* seleccionada.

De esta manera, para hallar el vector de movimiento, se escoge la *nube* que permite estimar el movimiento y se busca la misma *nube* en el *frame* siguiente. Así, el vector de movimiento será el que indica el cambio de coordenadas que sufre el centro de gravedad de dicha *nube* de un *frame* a otro. Según los estudios realizados, la *nube* escogida debe cumplir los siguientes requisitos:

- Debe contener un número medio de hops. Anteriormente, ya se ha comentado que una nube con pocos hops puede indicar, simplemente, que existe ruido en esos píxeles de la imagen y, por tanto, no sirve para estimar el movimiento. Por otro lado, elegir una nube que contiene muchos hops no es tampoco una solución acertada ya que las nubes muy grandes, al igual que ganan hops por un lado, pueden perderlos por otro de forma que su centro de gravedad baila respecto al centro del bloque. De esta manera, tampoco es posible estimar el movimiento.
- No debe tocar los bordes del bloque. Incluso debe existir cierta distancia a los mismos, de modo que la *nube* se pueda mover hacia ellos.
- Debe tener el mismo tipo de *hop* y contener un número de *hops* parecido que la nube del fotograma anterior que se ha escogido para la estimación del movimiento. Se trata de encontrar la misma nube y ver cómo se ha movido de un *frame* al siguiente.
- Las *nubes* de ambos *frames* deben tener una forma similar. Es decir, ambas deben poder encajar en un rectángulo imaginario de iguales dimensiones.
- Si existen varias *nubes* que cumplen las características anteriores, se supondrá que la válida es la más cercana a la que se ha escogido en el *frame* anterior.

Se ha de tener en cuenta que, en ocasiones, no existirán nubes que cumplan tales características. En estos casos existen dos soluciones:

- Considerar que el vector de movimiento es el mismo que ha sido aplicado en el *frame* anterior.
- No estimar ningún vector y enviar los T-hops que resultan de la primera codificación con LHE.

Según los estudios realizados, la mejor estrategia en estos casos en los que no se ha encontrado una *nube* adecuada es aplicar el vector generado en el *frame* anterior y calcular los T-hops utilizando como referencia el frame i al que se le ha aplicado el vector de movimiento (lo que da lugar a una aproximación

del frame i+1) y el frame original i+1. Si esta codificación contiene pocos hops grandes h_{-4} y h_4 significa que la aproximación de la estimación de movimiento es lo suficientemente buena como para enviar únicamente dicho vector en lugar de enviar la codificación de los T-hops.

Pese a todos los estudios realizados, este sistema aún se encuentra en fase experimental y será necesario evaluar otras estrategias basadas en *nubes* hasta llegar a una solución completamente satisfactoria.

6.6 Arquitectura

Algo a considerar como punto de partida es que la imagen se divide en una malla de bloques y cada bloque se evalúa de forma independiente. Así, se identifica el movimiento de las *nubes* de *hops* de cada bloque y se codifica la información resultante tras aplicar el vector de movimiento hallado.

Anteriormente se ha visto que para poder hallar los T-hops son necesarios dos fotogramas: uno de referencia y el que se quiere codificar. De esta manera, se usa como referencia un fotograma ficticio en el que todos los píxeles tienen el valor de luminancia Y=128. Los *T-hops* de los frames se calculan, por tanto, utilizando este *frame* de referencia y el *frame* que se desea codificar.

De esta manera, se realizan dos codificaciones: una para hallar los T-hops del *frame* i y otra para hallar los del *frame* i+1. Utilizando cada una de estas codificaciones, se realiza la estimación del movimiento tal y como se ha explicado en el apartado 6.5 Estimación del vector de movimiento basado en nubes. Finalmente, se codifica la Información Residual existente entre el *frame* que se obtiene al aplicar el vector de movimiento al *frame* i (lo que resulta en una aproximación del frame i+1) y el frame i+1. Si la Información Residual contiene muchos hops grandes h_{-4} y h_4, el vector de movimiento no ha sido bien estimado por lo que se codifican y envían los *T-hops* resultantes de la primera codificación LHE, en la que se tomó como referencia el *frame* cuyos píxeles tenían un valor de luminancia Y=128. Si, por el contrario, la Información Residual no contiene estos *hops* grandes, se codificarán y enviarán el vector de movimiento y los *T-hops* residuales, que permiten refinar aún más la imagen. Por otro lado, en el caso del primer *frame* de la película de video, se envían también los *T-hops* de la primera codificación LHE puesto que no existe un *frame* anterior sobre el que referenciar el movimiento.

En la Figura 54 se muestra un diagrama con la arquitectura del sistema anteriormente explicado.

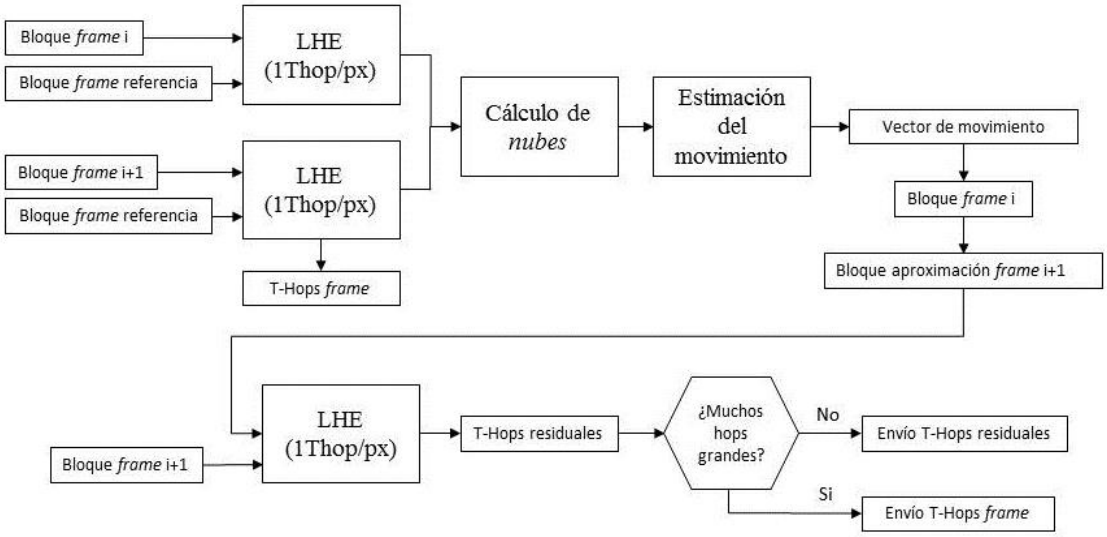


Figura 55. Arquitectura vídeo LHE

El Sistema anterior se puede representar, arquitecturalmente, como una codificación en dos pasos en la que como *input* se tiene un *frame* y su *frame* anterior y, como salida, los *T-hops* finales que se van a almacenar o enviar. Tras el primer LHE, se guardan las nuevas *nubes* obtenidas para comparar con las que se obtengan en el siguiente *frame*. La Figura 55 representa esta idea.

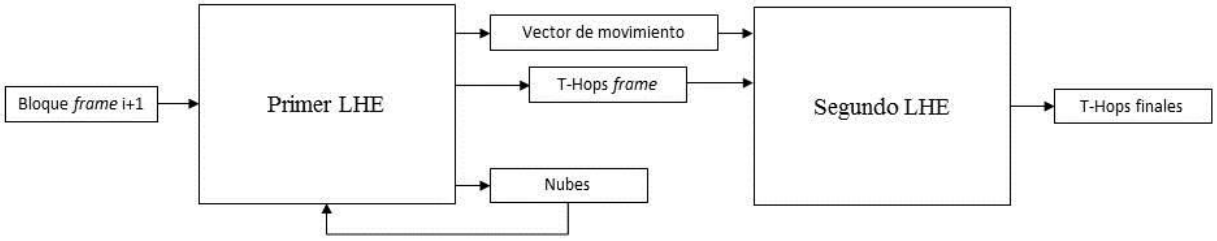


Figura 56. Arquitectura simplificada

7 Líneas futuras de investigación

Aunque, tal y como muestra esta memoria, se ha avanzado en las investigaciones con LHE, aún se puede continuar desarrollando y mejorando este algoritmo. De esta forma, las líneas de trabajo futuro son las siguientes:

- Mejora de la predicción de luminancia: estudiar otros métodos de predicción de la luminancia del píxel de forma que pudiésemos elegir siempre el *hop* nulo y los *hops* más pequeños h_{-1} y h_1. De esta manera, la imagen ocuparía menos bits por píxel y la relación de lo que ocupa la imagen con el PSNR obtenido sería mucho mejor que la actual.
- Mejora de *k(x)* por píxel: tal y como se explicó en el apartado 3.4.2.1 Valor de *k(x)* para cada píxel, los resultados de utilizar k(x) por píxel no son mejores que los que se obtienen utilizando k(x) por bloque. Sin embargo, ajustando este valor correctamente a cada píxel, el algoritmo se simplifica y se obtienen mejores tiempos de ejecución por lo que es necesario investigar más a fondo esta técnica de forma que se consigan buenos resultados en calidad.
- Mejora de las métricas de relevancia Perceptual: el objetivo es mejorar el *downsampling* realizado y, por tanto, la escalabilidad del algoritmo y la calidad de las imágenes. Aunque se obtienen resultados más que aceptables con el algoritmo actual, en algunas imágenes se producen saltos bruscos de calidad aunque se estén reduciendo (o aumentando) muy poco los bits por píxel. Con esta mejora se conseguirían resultados mucho mejores ya que la calidad no caería, tal y como ocurre en las gráficas Rate-Distorsion que se muestran en el apartado de resultados, sino que la relación bits por píxel- PSNR se mantendría de forma lineal.
- Compresión sin pérdidas: modificar el algoritmo para conseguir una compresión sin pérdidas. Se añadirían nuevos *hops* de forma que se pudiese codificar de forma más precisa el error cometido al realizar la predicción de la componente de color de un píxel. Estos saltos más finos ocuparían más bits y, por tanto, la compresión sería menor. La estrategia a seguir para estos *hops* más precisos sería realizar varias codificaciones de *hops* logarítmicos hasta aproximarse al brillo original. En la primera codificación puede cometerse un error, pero en la segunda, los *hops* tendríaN un rango más pequeño, concretamente el rango que estaba cubriendo el *hop* escogido en la primera codificación. Muchos píxeles se conseguirían con solo dos *hops* y alguno tendrían que usar tres *hops*. De esta manera, las imágenes pasarían a ocupar aproximadamente el doble pero la calidad obtenida también sería mucho mayor.
- Aplicar el algoritmo a archivos de música y estudiar los resultados.

- Mejora de vídeo: el estudio de vídeo ha comenzado con la estimación de vectores de movimiento. Sin embargo, aún queda mucho que explotar en este campo ya que servicios de vídeo interactivo tales como el cloud gaming podrían reemplazar el códec H.264 a favor de LHE si se llega a desarrollar completamente. El objetivo es conseguir *frames* de mejor calidad en un tiempo de codificación mucho menor de forma la experiencia de usuario pueda mejorar notablemente con respecto a la actual. Para ello es necesario:
    - Realizar pruebas masivas de robustez en el cálculo de vectores
    - Compresión de la información residual y evaluación de diferentes estrategias de *downsampling* tal y como se hace en la compresión de la imagen fija. Esto es posible ya que los *T-hops* tienen una altísima redundancia espacial, mayor aun que los *hops* espaciales, por lo que en principio deberían ser fácilmente compresibles con técnicas de escalado (*downsampling*) similares.
    - Evaluación de la calidad del vídeo generado con herramientas conocidas.

8 Conclusiones finales

Una de las conclusiones más importantes de este documento es que siempre se pueden desarrollar nuevas tecnologías que abran nuevos caminos y posibilidades de investigación. Es decir, no hay que cerrarse a desarrollar una idea simplemente porque la tecnología vigente va en otra dirección. De hecho, con este proyecto se ha demostrado que es posible trabajar en el dominio del espacio para realizar compresión de imagen y, además, lograr buenos resultados.

Para llegar a este punto, se han ido implementado diferentes versiones del algoritmo y haciendo pruebas sobre las mismas. De esta manera, se corregían los errores detectados en cada una de las versiones y se introducían nuevas mejoras. La última implementación realizada, que es la que se muestra en este documento, tiene como ventajas:

- Baja complejidad computacional: al trabajar en el dominio del espacio, LHE no requiere costosos cambios al dominio de la frecuencia por lo que la compresión es muy rápida.
- LHE ofrece una muy buena calidad subjetiva de la imagen, tal y como se ha visto en el apartado 5 Resultados, puesto que los errores que se cometen en el proceso de codificación tienen lugar en las áreas de la imagen que son menos relevantes para el observador.
- LHE permite comprimir imágenes a bajos *bit-rates* con una calidad realmente buena, mucho mejor que la que ofrece JPEG, tal y como se ha visto en el apartado 5 Resultados.
- En cuanto al sistema LHE para vídeo, aunque aún se encuentra en una fase de desarrollo bastante temprana, permitirá mejorar la experiencia de usuario en sistemas en los que los tiempos de latencia son vitales como, por ejemplo, en cloud gaming. Esto es debido a que su baja complejidad computacional y la posibilidad de hallar vectores de movimiento sin necesidad de realizar convoluciones permite que los tiempos de codificación y decodificación de vídeo sean más rápidos.

Debido a todas estas ventajas, es esencial divulgar este algoritmo de forma que otros puedan utilizarlo e, incluso, mejorarlo aportando nuevas ideas. Por esta razón, se pretende publicar un artículo que tiene como objetivo divulgar el conocimiento adquirido a la comunidad científica. Dicho artículo se incluye en el Anexo II.

Por último y de lo que no cabe duda es que LHE es una tecnología totalmente nueva y que acaba de nacer por lo que aún le queda mucho por crecer y mejorar.

9 Bibliografía

[1] Rabbani, M., Jones, P. W.: 'Adaptive DPCM', in 'Digital Image Compression Techniques' (SPIE Press, 1991)

[2] Gómez V., Juan Diego.: 'Compresión de Imágenes Digitales utilizando Redes Neuronales: Multicapa (Backpropagation) y RBR's'.

[3] Departamento de Informática, Universidad Técnica Federico Santa María: 'Algoritmos de interpolación de imágenes'.

[4] 'Kodak Lossless True Color Image Suite', http://r0k.us/graphics/kodak/, accedido January 2014

[5] 'USC-SIPI Image Database', http://sipi.usc.edu/database/database.php?volume=misc, accedido January 2014

[6] Lloyd, S.: 'Measures of complexity: a nonexhaustive list', IEEE Control Systems Magazine, 2001, 21, (4), pp. 7-8

[7] Jayant, N., Johnston, J., Safranek, R.: 'Signal compression based on models of human perception', Proceedings of the IEEE, Oct 1993, vol.81, no.10, pp.1385-1422

[8] Cruz, D. S., Ebrahimi, T., Larsson, M., Askelof, J., Cristopoulos, C.: 'Region of Interest coding in JPEG2000 for interactive client/server applications', IEEE 3rd Workshop on Multimedia Signal Processing, 1999, pp. 389-394

[9] Hassan, S.A., Hussain, M.: 'Spatial domain lossless image data compression method', 2011 International Conference on Information and Communication Technologies (ICICT), July 2011, pp.1-4

[10] Hasan, M., Nur, K., Noor, T. B., Shakur, H. B.: 'Spatial Domain Lossless Image Compression Technique by Reducing Overhead Bits and Run Length Coding', International Journal of Computer Science and Information Technologies (IJCSIT), 2012, vol. 3, no. 2

[11] Hasan, M., Nur, K.: 'A Novel Spatial Domain Lossless Image Compression Scheme', International Journal of Computer Applications, February 2012, vol. 39, no. 15, p. 25-28

[12] Huang, S. C., Chen, L. G., Chang, H. C.: 'A novel image compression algorithm by using Log-Exp transform', ISCAS '99. Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, Jul 1999, vol.4, pp.17-20

[13] Weinberger, M.J., Seroussi, G., Sapiro, G.: 'The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS'. IEEE Transactions on Image Processing, Aug 2000 vol.9, no.8, pp.1309-1324

[14] Christopoulos, C. A., Askelof J., Larsson, M.: ’Efficient Methods for Encoding Regions of Interest in the Upcoming JPEG 2000 Still Image Coding Standard’, IEEE Signal Processing Letters, September 2000, Vol. 7, No. 9, pp. 247-249

[15] Moreno, J.M.: ‘Perceptual Criteria on Image Compression’, Ph.D. dissertation, DCC, UAB, Barcelona, Spain, 2011

[16] Zhang, C. N., Wu, X.: ‘A hybrid approach of wavelet packet and directional decomposition for image compression’, IEEE Canadian Conference on Electrical and Computer Engineering, Edmonton, Alta., Canada, Dec. 1999, vol. 2, pp. 755-760

[17] Zhou, Z., Venetsanopoulos, A. N.: ‘Morphological methods in image coding’, IEEE International Conference on Acoustics, Speech, and Signal Process. (ICASSP-92), San Francisco, CA, USA, March 1992, vol. 3, pp. 481-484

[18] Popovici, I., Withers, W.D.: ‘Locating Edges and Removing Ringing Artifacts in JPEG Images by Frequency-Domain Analysis’, IEEE Transactions on Image Processing, May 2007, vol.16, no.5, pp.1470-1474

[19] ITU-T P.910: ‘Subjective Video Quality Assessment Methods for Multimedia Applications’, 2008

[20] Kok, C. W.: ‘Fast algorithm for computing discrete cosine transform’, IEEE Transactions on Signal Processing, 1997, 45, (3), pp 757-760

[21] Rafael C. González, Richard E. Woods: ‘Digital Image Processing’.

10 ANEXO I: CÓDIGO

10.1 Común

10.1.1 Fichero

- BufferedImage loadImage(String pathImagen)

```
/**
 * Carga una imagen procedente del sistema localizada en el directorio pathImagen
 * @param pathImagen Path a la imagen que se desea cargar
 * @return img Imagen cargada en BufferedImage
 */
public BufferedImage loadImage(String pathImagen) {

        BufferedImage img=null;

        try {
            img = ImageIO.read(new File(pathImagen));
        } catch (IOException e) {
                System.out.println("No se ha conseguido cargar la imagen");
                System.exit(0);
        }

        return img;
}
```

- int[] imgToInt(BufferedImage img)

```
/**
 * Una vez cargada transforma el objeto img en un array que contiene los valores de la componente de
color de cada píxel.
 * @param img imagen
 * @return array con las luminancias
 */
public int[] imgToInt(BufferedImage img) {
         int ancho=img.getWidth(); ANCHO=ancho;
         int alto=img.getHeight();  ALTO = alto;

         int[] luminancia=new int[ancho*alto];
         int i=0;
         for (int y=0;y<alto;y++)  {
                 for (int x=0;x<ancho;x++)  {
                         int c=img.getRGB(x, y);

                         int red=(c & 0x00ff0000) >> 16;
                         int green=(c & 0x0000ff00) >> 8;
                         int blue=(c & 0x000000ff);

                         luminancia[i]=(red*299+green*587+blue*114)/1000; //funcion de luminancia

                         i++;
                 }
         }
return luminancia;
}
```

- boolean salvarImagen(final BufferedImage imagen, final String rutaFichero, final String formato)

```
public boolean salvarImagen(final BufferedImage imagen,
                        final String rutaFichero, final String formato) {

        String ruta = rutaFichero + "." + formato;

        // Comprobacion de parametros
        if (imagen != null && rutaFichero != null && formato != null) {

                try {
                        ImageIO.write( imagen, formato, new File(ruta));
                        return true;
                } catch (Exception e){
                        System.out.println("No se ha podido generar el archivo");// Fallo al guardar
                }
                        return false;
                } else {
                        // Fallo en los parametros
                        return false;
                }
        }
```

- float saveLHEFile (String pathFile, int ancho, int alto, int lum, int cromU, int cromV, String mode, String malla, String image, String image_cromU, String image_cromV, boolean codificado)

```
public float saveLHEFile (String pathFile, int ancho, int alto, int lum, int cromU, int cromV,  String
mode, String malla, String image, String image_cromU, String image_cromV,  boolean codificado) {
        boolean crominancia= true;

        if (mode.equals("")) crominancia = false;

        String ancho_bits =  Integer.toString(ancho, 2);
        String alto_bits = Integer.toString(alto, 2);
        String lum_bits = Integer.toString(lum, 2);
        String cromU_bits = Integer.toString(cromU, 2);
        String cromV_bits = Integer.toString(cromV, 2);

        StringBuilder cabecera = new StringBuilder(""); //La cabecera
        StringBuilder archivo = new StringBuilder ("");

        //Bits para indicar si es B/N o modo de crominancia 420 o 422
        if (mode.equals("")) {
                cabecera = cabecera.append("00");
        } else if (mode.equals("420")) {
                cabecera = cabecera.append("01");
        } else if (mode.equals("422")) {
                cabecera = cabecera.append("10");
        }
```

```
// Se ajustan las longitudes de las diferentes partes de la cabecera
        if (ancho_bits.length()%16!= 0) {
                for (int i=0; i<(ancho_bits.length()%16);i++) {
                        ancho_bits = "0".concat(ancho_bits);
                }
        }

        if (alto_bits.length()%16!= 0) {
                for (int i=0; i<(alto_bits.length()%16);i++) {
                        alto_bits = "0".concat(alto_bits);
                }
        }

        if (lum_bits.length()%8!= 0) {
                for (int i=0; i<(lum_bits.length()%8);i++) {
                        lum_bits = "0".concat(lum_bits);
                }
        }

        if (cromU_bits.length()%8!= 0) {
                for (int i=0; i<(cromU_bits.length()%8);i++) {
                        cromU_bits = "0".concat(cromU_bits);
                }
        }

        if (cromV_bits.length()%8!= 0) {
                for (int i=0; i<(cromV_bits.length()%8);i++) {
                        cromV_bits = "0".concat(cromV_bits);
                }
        }

        //Bit que indica si se ha usado Huffman o no
        String bitHuffman="";
        if (hayHuffman) bitHuffman = "1";
        else bitHuffman = "0";

        //Enviamos la tabla Huffman si ha habido Huffman
        StringBuilder tablaHuffman = new StringBuilder("");
        if (hayHuffman) {
                for (int i=1; i<huffman.length; i++) {
                        if (huffman[i]!=null) {
                                String longitud = Integer.toString(huffman[i].length(), 2);
                                if (longitud.length()%4!= 0) {
                                        for (int j=0; j<(longitud.length()%4);j++) {
                                                longitud = "0".concat(longitud);
                                        }
                                }
                                tablaHuffman = tablaHuffman.append(longitud).append(huffman[i]);
                        } else {
                                tablaHuffman = tablaHuffman.append("0000"); //Si algún código no se ha
usado se indica que la longitud es cero, y no hay código
                        }
                }
        }

        StringBuilder kopt = new StringBuilder("");

        for (int i=0; i<kguay.size(); i++) {
                String k = kguay.get(i).toString();
                switch (k) {
                        case "3.0":
                                kopt = kopt.append("0");
                                break;
                        case "4.0":
                                kopt = kopt.append("1");
                                break;
```

```
                        default:
                                System.out.println("ERROR");
                                break;
                }
        }

        if                          (crominancia)                          cabecera                          =
cabecera.append(ancho_bits).append(alto_bits).append(lum_bits).append(cromU_bits).append(cromV_bits).appe
nd(bitHuffman).append(tablaHuffman).append(malla).append(kopt)
        else                                          cabecera                                                  =
cabecera.append(ancho_bits).append(alto_bits).append(lum_bits).append(bitHuffman).append(tablaHuffman).ap
pend(malla).append(kopt)

        if (archivo.length()%8!= 0) {
                for (int i=0; i<(archivo.length()%8);i++) {
                        archivo = archivo.append("0");
                }
        }

        //Creamos un vector de string que almacenará los bytes a guardar
        Vector<String> segment = new Vector<String>();

        //Cada elemento del vector será un string que representa 8 bits para que así podamos guardar byte
a byte
        for (int i=0; i<archivo.length();) {
                segment.add(archivo.substring(i, i+8));
                i+=8;
        }
        //Generamos el fichero LHE, solo si se corresponde con los bpp que se desean
        if (codificado) {
                try{
                        DataOutputStream   dos   =   new   DataOutputStream(new   BufferedOutputStream(new
                FileOutputStream(pathFile)));
                        /* Escribimos byte a byte */
                        for(int i=0;i<segment.size();i++){
                                Byte b = (byte)Short.parseShort(segment.get(i), 2);
                                dos.writeByte(b);
                        }

                        dos.close();
                } catch (Exception e){
                        System.out.println("No se ha podido generar el archivo");// Fallo al guardar
                }
        }
```

- String mallasToBits (Vector<Integer> malla)

```
/**
* Codifica las mallas
* @param malla
* @return cadena String con los bits codificados
*/
public String mallasToBits (Vector<Integer> malla) {
        //Hacemos una estadistica para ver qué escale_type tenemos
        int [] estadistica = new int [8];
        for (int i=0; i<malla.size();i++) {
                estadistica[malla.get(i)]++;
        }
        //Inicializamos
        for (int i=0; i<orden.length;i++) {
                orden [i] = i;
        }
        //Ordenamos de menor a mayor
        int aux = 100;
```

```
	for (int i=0; i<estadistica.length-1; i++) {
		for (int x=i+1; x<estadistica.length; x++) {
			if(estadistica[x]<estadistica[orden[i]]) {
				aux = orden[i];
					orden [i]= orden[x];
					orden [x] = aux;

			}
		}
	}


	//Las mallas se codificarán:
	// 1. Indicando el orden en el que se han encontrado los escale_type
	// 2. Codificando con el código Huffman correspondiente.

	StringBuilder mallaBits=new StringBuilder("");

	// Codificamos para enviar el orden que tendrán las mallas

	for (int k=orden.length-1; k>=0; k--) {
		int num = orden[k];
		switch (num) {
			case 0:
				mallaBits = mallaBits.append("000");
				break;
			case 1:
				mallaBits = mallaBits.append("001");
				break;
			case 2:
				mallaBits = mallaBits.append("010");
				break;
			case 3:
				mallaBits = mallaBits.append("011");
				break;
			case 4:
				mallaBits = mallaBits.append("100");
				break;
			case 5:
				mallaBits = mallaBits.append("101");
				break;
			case 6:
				mallaBits = mallaBits.append("110");
				break;
			case 7:
				mallaBits = mallaBits.append("111");
				break;
		}

	}

	for (int i=0; i<malla.size();i++) {
		int valor = malla.get(i);
		if (valor == orden[7]) {
			mallaBits = mallaBits.append("1");
		} else if (valor == orden[6]) {
			mallaBits = mallaBits.append("01");
		}  else if (valor == orden[5]) {
			mallaBits = mallaBits.append("001");
		} else if (valor == orden[4]) {
			mallaBits = mallaBits.append("0001");
		} else if (valor == orden[3]) {
			mallaBits = mallaBits.append("00001");
```

```
                } else if (valor == orden[2]) {
                        mallaBits = mallaBits.append("000001");
                } else if (valor == orden[1]) {
                        mallaBits = mallaBits.append("0000001");
                } else if (valor == orden[0]) {
                        mallaBits = mallaBits.append("0000000");
                } else {
                        System.out.println("Ha habido un error en la cadena");
                }

        }

        return mallaBits.toString();

}
```

- String imageToBits (PerceptualRelevance pr, String modo, **int**[] comp_image, **boolean** codigo_huff)

```
public String imageToBits (PerceptualRelevance pr, String modo, int[] comp_image, boolean codigo_huff) {
        //antes de entrar aqui hay que llamar a computeBppHop() y computeBppTr()
        int alto=ALTO;
        int ancho=ANCHO;
        if (modo.equals("")) hayHuffman = codigo_huff; // solo para luminancia

        StringBuilder image= new StringBuilder("");

        int total_tamano=0;
        int bloques=0;
        if (pr.block_list.size()>0) bloques=pr.block_list.size();
        else bloques=1;


        int num_vacios=0;
        int ahorro_vacios=0;

        int [] codigos = new int [10];

        Vector<Integer> longitudes = new Vector<Integer>();

        for(int bi=0; bi<bloques; bi++) {
              PerceptualRelevance.Block b= null;
              if (pr.block_list.size()>0) b=pr.block_list.get(bi);
              else {
                  PerceptualRelevance praux=new PerceptualRelevance();
                  praux.ancho=ancho;
                  praux.alto=alto;

                   b=praux.new Block(0,0, //origen
                                     ancho-1,alto-1,//fin
                                ancho-1,alto-1,//escaled (no hay escalados)
                                1, //type=1;
                                0, //malla0
                                0,"");//indice de bloque

                   pr=praux;
         }

         if (modo.equals("")) setBppNoScaled();
         else setBppColor();
```

```
int tamano_b=0;//tamano del bloque en bits codificados

boolean vacio=true;

int xfin_scaled = b.xfin_scaled;
int yfin_scaled = b.yfin_scaled;

if (modo.equals("420")) {
        xfin_scaled = b.xini + (b.xfin_scaled-b.xini)/2;
        yfin_scaled = b.yini + (b.yfin_scaled-b.yini)/2;
 } else if (modo.equals("422")) {
        xfin_scaled = b.xini + (b.xfin_scaled-b.xini)/2;
 }

  StringBuilder bloque = new StringBuilder ("");
  Vector<Integer> longitudesBloque = new Vector<Integer>();

  for (int y=b.yini;y<=yfin_scaled;y++){
           for (int x=b.xini;x<=xfin_scaled;x++) {
                int hoplen=100;//bpp_hop[h];//a es lo que ocuparia el simbolo
                int ahorro=0; //posibles bits de ahorro
                int hop_ant=-1; //hop anterior (en x-1) , si existe
                int up=-1;//simbolo estimado por up (osea simbolo up)
                int sym = -1;// simbolo estimado por tr1


                int pos_b= y*ancho+x;
                int h=comp_image[pos_b];

                if (x>0) hop_ant= comp_image[pos_b-1];
                if (hop_ant==0) hop_ant=-1;

                int upi=-1;
                int pos_upi=(y-1)*pr.ancho+x-1;
                if ((x>b.xini) && y>b.yini) upi=comp_image[pos_upi];
                if (upi==0) upi=-1;

                int pos_up=  (y-1)*pr.ancho+x;
                if (y>0) up=comp_image[pos_up];
                if (up==0) up=-1;

                sym = bpp_tr_predice[h][1];

                 hoplen=bpp_hop[h].length();

                //============================================================
                //el concepto de caso es una configuracion de orden, nada mas
                int[] orden_hops=new int[10];


                int um=5;
                if (up!=-1 && up<um) sym=up;
                else if (upi!=-1 && upi<um) sym=upi;

                orden_hops[0]=sym;
                orden_hops[1]=up;
                orden_hops[2]=hop_ant;
                orden_hops[3]=-1;
                orden_hops[4]=-1;
                orden_hops[5]=-1;
                orden_hops[6]=-1;
                orden_hops[7]=-1;
                orden_hops[8]=-1;
                orden_hops[9]=-1;
```

```
                    HashSet descartados=new HashSet();//lista de hops descartados
                    boolean encontrado=false;
                    int tam_pix=0;
                    for (int k=0;k<5;k++) {
                            int ho=orden_hops[k];
                            if      ((ho!=-1)       &&      (!descartados.contains(ho)))      {

                                    if (h==ho) {
                                            tamano_b+=1;
                                            tam_pix++;
                                            encontrado=true;
                                            if (!codigo_huff) bloque = bloque.append("1");
                                            descartados.add(ho);
                                            break;//sale del for ho
                                    } else { //el hop no coincide
                                            Vacio=false;
                                            tamano_b+=1;// hop q no esta descartado
                                            if (!codigo_huff) bloque =bloque.append("0");
                                            tam_pix++;
                                            if (bpp_hop[ho].length()<hoplen) ahorro+=1;
                                            descartados.add(ho);

                                    }//else
                            }//-1
                     }// for k hops ordenados

                    if (tam_pix>1) vacio=false;

                    if (!encontrado) {
                    tamano_b+=hoplen-ahorro;
                    tam_pix+=hoplen-ahorro;
                    //Se genera el código
                    if (!codigo_huff) {
                            String simbolo ="";
                            simbolo = bpp_hop[h].substring(ahorro);
                            bloque =bloque.append(simbolo);
                    }
            }

            codigos[tam_pix]++;
            longitudesBloque.add(tam_pix);

        }//x
}//y
if (!modo.equals("")) vacio = false; // Para el color no se contemplan vacíos

if (vacio==true) {
        pr.block_list.get(bi).scale_type+=4; // los scale_type de vacio comienzan en 4
        pr.escale_type[b.malla] [b.block_index] +=4; //también en el array scale_type
        num_vacios++;
        ahorro_vacios+=tamano_b;//anchobx*anchoby es lo mismo que tamano_b

} else {
        image = image.append(bloque);
        longitudes.addAll(longitudesBloque);
}

total_tamano+=tamano_b;//+ bits_parada;

}//end for block
```

```
        //vamos a hacer una traduccion a codigo huffman. tabla unica para toda la imagen
        //-------------------------------------------------------
        if (codigo_huff) {
                StringBuilder imageHuff = new StringBuilder("");
                Huffman huff =new Huffman();
                huffman=huff.getTranslateCodesString(codigos);

                for (int i=0; i<longitudes.size(); i++) {
                        imageHuff = imageHuff.append(huffman[longitudes.get(i)]);
                }

                    int tamano=ancho*alto;

                    return imageHuff.toString();

        }
        //-------------------------------------------------------------------------

        int tamano=ancho*alto;
        float k=(float)total_tamano/(float)tamano;
        return image.toString();

}
```

- PerceptualRelevance readLHEFile (String pathFile)

```
/**
* Decodifica archivo LHE
* @param pathFile path del archivo
* @return objeto PerceptualRelevance con la info para decodificar
*/
public PerceptualRelevance readLHEFile (String pathFile) {
        //Lectura del archivo
        StringBuilder archivo = new StringBuilder ("");
        Byte b;
        File file = new File(pathFile);
        int len=(int)file.length();

        int actualBit=0;
        int cuentaLeeBits=0;

        try{
            DataInputStream      dis      =      new      DataInputStream(new      BufferedInputStream(new
FileInputStream(pathFile)));

        /* Leemos byte a byte */
        while (len>0) {
        len--;
        b = dis.readByte();
        archivo   =   archivo.append(String.format("%8s",Integer.toBinaryString(b   &   0xFF)).replace('   ',
'0'));
        }
        dis.close();

        } catch (Exception e){
                System.out.println("No se puede leer el archivo");// Fallo al leer
        }
```

```
        length = archivo.length();
        //Parámetros de configuración del decoder
        int num_mallas=3;
        int num_hops=8;
        int block_len_min=8;
        cuentaLeeBits+=2;
        //Sacamos si la imagen está en B/N o a color y su modo de crominancia
        String modeBits = archivo.substring(actualBit,cuentaLeeBits);
        if (modeBits.equals("00")) {
                modoCrominancia = "";
        } else if (modeBits.equals("01")) {
                modoCrominancia ="420";
        } else if (modeBits.equals("10")) {
                modoCrominancia="422";
        }

        actualBit= cuentaLeeBits;

        boolean crominancia = true;
        if (modoCrominancia.equals("")) crominancia = false;
        cuentaLeeBits+=16;
        //Sacamos el ancho
        String ancho_bits = archivo.substring(actualBit, cuentaLeeBits);
        int ancho = Integer.parseInt(ancho_bits, 2);
        ANCHO = ancho;
        actualBit= cuentaLeeBits;

        cuentaLeeBits+=16;
        //Sacamos el alto
        String alto_bits = archivo.substring(actualBit, cuentaLeeBits);
        int alto = Integer.parseInt(alto_bits, 2);
        ALTO = alto;
        actualBit= cuentaLeeBits;


        // Generamos objeto Perceptual Relevance
        PerceptualRelevance   prdec=new   PerceptualRelevance(ANCHO,   ALTO   ,   num_hops,   num_mallas,
block_len_min);
        // Inicializamos los arrays
        luminancia = new int[ANCHO*ALTO];
        crominanciaU = new int[ANCHO*ALTO];
        crominanciaV = new int[ANCHO*ALTO];

        //Sacamos la luminancia y crominancia escaled
        cuentaLeeBits+=8;
        String lum_bits = archivo.substring(actualBit, cuentaLeeBits);
        int lum = Integer.parseInt(lum_bits, 2);
        prdec.lum_escaled[0] = lum;
        actualBit= cuentaLeeBits;

        if (crominancia) {
                cuentaLeeBits+=8;
                String cromU_bits = archivo.substring(actualBit, cuentaLeeBits);
                int cromU = Integer.parseInt(cromU_bits, 2);
                prdec.crominanciaU_escaled[0] = cromU;
                actualBit= cuentaLeeBits;

                cuentaLeeBits+=8;
                String cromV_bits = archivo.substring(actualBit, cuentaLeeBits);
                int cromV = Integer.parseInt(cromV_bits, 2);
                prdec.crominanciaV_escaled[0] = cromV;
                actualBit= cuentaLeeBits;
        }
```

```
        cuentaLeeBits+=1;
        //Bit que indica si ha habido Huffman o no
        String bitHuffman = archivo.substring(actualBit,cuentaLeeBits);
        if (bitHuffman.equals("1")) hayHuffman = true;
        else if (bitHuffman.equals("0")) hayHuffman = false;
        actualBit= cuentaLeeBits;


        int cuenta = 0;
        cuentaLeeBits+=4;
        if (hayHuffman) {
                for (int i=1; i<huffman.length;i++){
                        String bits_long = archivo.substring(actualBit+cuenta, cuentaLeeBits+cuenta);
                        int longitud = Integer.parseInt(bits_long, 2);
                        if (longitud==0) cuenta+=4;
                        else if (longitud>0) {
                        huffman[i]                =                archivo.substring(cuentaLeeBits+cuenta,
cuentaLeeBits+cuenta+longitud);
                        cuenta+=4+longitud; //los 4 bits que ocupa la longitud y la longitud
                        }
                }
        }

        actualBit+=cuenta;
        StringBuilder mallasYsimbolos = new StringBuilder(archivo.substring(actualBit));

        //Variables para leer mallas, bucle while hasta que se acaben
        boolean finMalla=false;
        leidos = 0; //Llevará la cuenta de los bits que se han leído del fichero
        malla=2; //Cuando se codifica, se empieza siempre por la malla2. También al decodificar

        contador = new int[prdec.num_mallas]; // Contadores de bloques de mallas
        int [] nBloquesHMalla = new int [prdec.num_mallas]; // Bloques en horizontal de cada malla
        for (int i=0; i<prdec.num_mallas; i++) {
                nBloquesHMalla[i]=ancho/prdec.block_len[i];
        }
        //Antes de generar los bloques se mira el orden que se ha seguido para saber el código Huffman
        String bits = "";
        for (int i=0; i<orden.length;i++) {
                bits = mallasYsimbolos.substring(leidos, leidos+3);
                orden[i] = Integer.parseInt(bits, 2);
                leidos+=3;
        }

        while (finMalla==false) {
                finMalla=    generaBlock(prdec,    mallasYsimbolos,    nBloquesHMalla,    ancho,    alto,
modoCrominancia);
        }

        k = new Vector<BigDecimal>();
        //Ya tenemos los bloques generados, ahora hay que rellenar el vector de K óptima
        for (int i=0; i<prdec.block_list.size(); i++) {
                bits = mallasYsimbolos.substring(leidos, leidos+1);
                switch (bits) {
                        case "0":
                                k.add(new BigDecimal("3.0"));
                                break;
                        case "1":
                                k.add(new BigDecimal("4.0"));
                                break;
                        default:
                                System.out.println("ERROOOR");
                }
                        leidos+=1;
        }
```

```
        generaHops (prdec, mallasYsimbolos, luminancia,  "" , hayHuffman);

        if (crominancia) {
                generaHops (prdec, mallasYsimbolos, crominanciaU, modoCrominancia, false);
                generaHops (prdec, mallasYsimbolos, crominanciaV, modoCrominancia, false);
        }
        return prdec;
}
```

10.1.2 PerceptualRelevance

- Constructores

```
/**
* Constructor vacío PerceptualRelevance
 */
public PerceptualRelevance() { }

/**
*  Constructor Encoder
* @param ancho
* @param alto
* @param num_hops
* @param num_mallas
* @param len_min
* @param calidad
*/
public PerceptualRelevance(int ancho, int alto,
                           int num_hops,      // esto es 8, o 6 o 4 o 2, o 10
                           int num_mallas,
                           int len_min,
                           float calidad
                           ){
        this.ancho=ancho;
        this.alto=alto;
        this.num_mallas=num_mallas;
        this.num_hops=num_hops;

        //Dimensionar los arrays
        lum_orig=new int[ancho*alto];
        lum_escaled=new int[ancho*alto];
        crominanciaU_escaled = new int[ancho*alto];
        crominanciaV_escaled = new int[ancho*alto];

        lum_rescaled =new int[ancho*alto];
        crominanciaU_rescaled =new int[ancho*alto];
        crominanciaV_rescaled =new int[ancho*alto];

        rescaled_hops = new int[ancho][alto];
        rescaled_hops_cromU = new int[ancho][alto];
        rescaled_hops_cromV = new int[ancho][alto];

        cache_blocks=new Block[ancho][alto];

        block_list=new ArrayList<Block>();
        block_len=new int[num_mallas];
        block_len[0]=len_min;
```

```
        //cada bloque de la siguiente malla contiene 4 subbloques.
        for (int i=1;i<num_mallas;i++) block_len[i]=block_len[i-1]*2;

        //Umbrales
        th_Savg_pelo=new float[this.num_hops+1][num_mallas];
        th_Savg_liso=new float[this.num_hops+1][num_mallas];
        th_Sh=new float[this.num_hops+1][num_mallas];
        th_Sv=new float[this.num_hops+1][num_mallas];

        escale_type=new int[num_mallas][ancho*alto/(len_min*len_min)];
        Savg=new float[num_mallas][ancho*alto/(len_min*len_min)];
        Sh=new float[num_mallas][ancho*alto/(len_min*len_min)];
        Sv=new float[num_mallas][ancho*alto/(len_min*len_min)];
        THconfigCalidad(calidad); // rellena los valores de los umbrales
}
/**
* Constructor Decoder
* @param ancho
* @param alto
* @param num_hops
* @param num_mallas
* @param len_min
*/
public PerceptualRelevance(int ancho, int alto,
            int num_hops,        // esto es 8, o 6 o 4 o 2, o 10
            int num_mallas,
            int len_min ){
        this.ancho=ancho;
        this.alto=alto;
        this.num_mallas=num_mallas;
        this.num_hops=num_hops;
        //Dimensionar los arrays
        lum_escaled=new int[ancho*alto];
        crominanciaU_escaled = new int[ancho*alto];
        crominanciaV_escaled = new int[ancho*alto];

        lum_encoded = new int[ancho*alto];
        crominanciaU_encoded = new int[ancho*alto];
        crominanciaV_encoded = new int[ancho*alto];

        lum_rescaled =new int[ancho*alto];
        crominanciaU_rescaled =new int[ancho*alto];
        crominanciaV_rescaled =new int[ancho*alto];

        lum_rescaled_final=new int[ancho*alto];
        crominanciaU_rescaled_final = new int[ancho*alto];
        crominanciaV_rescaled_final = new int[ancho*alto];

        rescaled_hops = new int[ancho][alto];
        rescaled_hops_cromU = new int[ancho][alto];
        rescaled_hops_cromV = new int[ancho][alto];

        pixelA=new Pixel[ancho][alto];
        pixelB=new Pixel[ancho][alto];
        pixelC=new Pixel[ancho][alto];
        pixelD=new Pixel[ancho][alto];

        cache_blocks=new Block[ancho][alto];
        block_list=new ArrayList<Block>();
        block_len=new int[num_mallas];
        block_len[0]=len_min;
        //cada bloque de la siguiente malla contiene 4 subbloques.
        for (int i=1;i<num_mallas;i++) block_len[i]=block_len[i-1]*2;
        escale_type=new int[num_mallas][ancho*alto/(len_min*len_min)];
}
```

- void THconfigCalidad(float calidad)

```
/**
* Inicializa umbrales según la calidad escogida y el número de mallas
*/
public void THconfigCalidad(float calidad) {
        factor=new float[3];
        factor[0]=1.5f;
        factor[1]=1.5f;
        factor[2]=1.5f;

        if (calidad==11) calidad=1000000;
        float q=(float)calidad;//1.0f-(float)Math.log((double)(calidad));

        double cf=(double) calidad*10;
        q = (float)(Math.exp(-0.50f*cf/10.0));

        if (calidad==-1) q=10000;//

        //si calidad es 10 es calidad maxima y q vale 0
        //liso significa que los simbolos son bajos o bien que son altos pero Sh y SV son casi cero
        th_Savg_liso[8][0]=q;//9f;//default 0.2
        //pelo es que los simbolos son altos, aunque si Sh y Sv son casi cero, es pelo.
        th_Savg_pelo[8][0]=Math.max (q+0.3f,1.0f);

        th_Sh[8][0]=Math.min(q/2.0f,0.3f);//8f;//default 0.1
        th_Sv[8][0]=Math.min(q/2.0f,0.3f);;//8f;//default 0.1

        //en malla 1 aplic4 3.5 veces los umbrales de malla 0, aunque podrian ser menores
        //sin embargo, ya ponerlos como 4 veces es exigir menor relevancia pues
        if (num_mallas>=2)  {
                th_Savg_liso[8][1]=0.8f*th_Savg_liso[8][0];//default 0.2
                th_Savg_pelo[8][1]=4f*th_Savg_pelo[8][0];//default 0.85

                th_Sh[8][1]=2f*th_Sh[8][0];//8f;//default 0.1
                th_Sv[8][1]=2f*th_Sv[8][0];//8f;//default 0.1

        }

        if (num_mallas>=3)  {
                th_Savg_liso[8][2]=0.7f*th_Savg_liso[8][1];//default 0.2
                th_Savg_pelo[8][2]=4*th_Savg_pelo[8][1];//default 0.85
                th_Sh[8][2]=1.5f*th_Sh[8][1];//8f;//default 0.1
                th_Sv[8][2]=1.5f*th_Sv[8][1];//8f;//default 0.1
        }

        if (num_mallas>=4) {
                th_Savg_liso[8][3]=0.6f*th_Savg_liso[8][2];//default 0.2
                th_Savg_pelo[8][3]=6*th_Savg_pelo[8][2];//default 0.85
                th_Sh[8][3]=1.5f*th_Sh[8][2];//8f;//default 0.1
                th_Sv[8][3]=1.5f*th_Sv[8][2];//8f;//default 0.1
        }

        if (num_mallas>=5)   {
                th_Savg_liso[8][4]=0.5f*th_Savg_liso[8][3];//default 0.2
                th_Savg_pelo[8][4]=6*th_Savg_pelo[8][3];//default 0.85
                th_Sh[8][4]=1.5f*th_Sh[8][3];//8f;//default 0.1
                th_Sv[8][4]=1.5f*th_Sv[8][3];//8f;//default 0.1
        }
}
```

- void computeMetrics(int macrobloque)

```
/**
* Calcula los valores a distintas escalas de las metricas de
* relevancia perceptual de un macrobloque, y con esos valores y
* los umbrales, determina los tipos de escalado a aplicar en cada nivel
* Genera una lista de subbloques
* metricas + umbrales-->tipos de escalados--->lista de subbloques
*
* @param macrobloque macrobloque del que se procesan los símbolos
*/
public void computeMetrics( int macrobloque) {
        //las metricas me las piden de un macrobloque (el mayor posible)
        //los simbolos los tenemos en el array hops

        //la malla cero es la menor. mide 8 ( default)
        //la malla num_mallas-1 es la mayor
        //el macrobloque esta referenciado a la malla mayor
        int bloque_malla_0=getB(macrobloque, num_mallas-1,0);
        int ancho_bloques=block_len[num_mallas-1]/block_len[0];
        int bloque_fin_0=bloque_malla_0+ancho_bloques-1+(ancho_bloques-1)*ancho/block_len[0];
        int bloques=block_len[num_mallas-1]/block_len[0];

        //ahora voy a recorrer cada bloque del macrobloque 3 veces para rellenar cada medida

        for  (int  bloque_ini=bloque_malla_0;bloque_ini<=bloque_fin_0;bloque_ini+=ancho/block_len[0])    {
                        for (int b=bloque_ini;b<bloque_ini+bloques;b++)   {
                        //la variable b contiene el indice de bloque en la malla 0
                        int[] xy=getxy(b,0);
                        //vamos a recorrer el bloque. La primera vez calculo Savg y Sv y luego calculo Sh
                        //asi solo recorro el bloque dos veces en lugar de 3
                        //inicializo las metricas del bloque b en malla 0
                        Sv[0][b]=0;Sv[0][b]=0;Savg[0][b]=0;

                        int signo_ant=0;
                        int signo=0;
                        for (int i=0;i<block_len[0];i++)
                                for (int j=0;j<block_len[0];j++)   {
                                        int x2=xy[0]+i;
                                        int y2=xy[1]+j;
                                        int pos=y2*ancho+x2;


                                        int s=hops[pos];
                                        if (s%2!=0) s+=1;//debe pesar lo mismo un negativo y un positivo
                                        Savg[0][b]+=s;

                                        signo= hops[pos]%2;
                                        if ((j==0) ) signo_ant=signo;
                                        if (signo!=signo_ant) Sv[0][b]++;//agitacion fuerte o debil
                                        signo_ant=signo;
                                }

                                //normalizo     Savg y Sv
                                Savg[0][b]=1.0f-((Savg[0][b])/(num_hops*block_len[0]*block_len[0]));
                                Sv[0][b]=Sv[0][b]/((block_len[0])*block_len[0]);
                                //ahora recorremos linea a linea en horizontal para calcular Sh
                                signo_ant=0;
```

```
                                for (int j=0;j<block_len[0];j++)
                                        for (int i=0;i<block_len[0];i++) {
                                                int x2=xy[0]+i;
                                                int y2=xy[1]+j;
                                                int pos=y2*ancho+x2;
                                                signo= hops[pos]%2;      //par =positivo, impar=negativo
                                                if ((i==0) ) signo_ant=signo;
                                                if (signo!=signo_ant) Sh[0][b]++;//agitacion
                                                signo_ant=signo;

                                        }
                                        //normalizo     Sh
                                        Sh[0][b]=Sh[0][b]/((block_len[0])*block_len[0]);

                                        //ahora que ya tengo las metricas puedo calcular los tipos de
                                        escalado
                                        //para ello comparo con los umbrales
                                        int tipo=1;//no clasificado, osea relevante

                                        if (Savg[0][b]<th_Savg_liso[num_hops][0]) tipo=0;//liso
                                        if (Savg[0][b]>th_Savg_pelo[num_hops][0]) tipo=5;//pelo

                                         //nuevo
                                float margen_v=th_Sv[num_hops][0]-Sv[0][b]; //positive si margen
                                float margen_h=th_Sh[num_hops][0]-Sh[0][b]; //positive si margen

                                if ((tipo==1) || (tipo==5)) {
                                        if (margen_v>0 && margen_v>margen_h)tipo=3;
                                else  if (margen_h>0 && margen_h>margen_v) tipo=2;
                                }

                                //el pelo se escala en direccion preferente
                                if (tipo==5) {
                                        if (Sv[0][b]<Sh[0][b]) tipo=3;
                                        else tipo=2;
                                }
                                //para luego poder comprobar si ha sido calculada
                                Savg[0][b]+=0.0001;

                                if (tipo==1) {//los tipos 1 gastan muchos hops, nada menos que 64 hops
                                        if (Savg[0][b]<2*th_Savg_liso[num_hops][0])
                                        if (margen_v>0 && margen_v>margen_h)tipo=3;
                                        else if (margen_h>0 && margen_h>margen_v)tipo=2;
                                        else {
                                            if (Sv[0][b]<Sh[0][b]) tipo=3;
                                            else tipo=2;
                                        }
                                 }

                                escale_type[0][b]=tipo;//
                        }//end for recorrer subbloques en horizontal
                }//end for recorrer bloques en vertical

                fillScaleType(num_mallas-1,macrobloque);

}//end funcion
```

- void escaleArea(int xini,int yini,int xfin, int yfin, int xfinscaled, int yfinscaled)

```
/**
* Esta función es de escalado, da mayor peso a los pixeles cercanos al pixel escalado
*
* @param xini coordenada x inicial del bloque
* @param yini coordenada y inicial del bloque
* @param xfin coordenada x final del bloque
* @param yfin coordenada y final del bloque
* @param xfinscaled coordenada x final del bloque escalado
* @param yfinscaled coordenada y final del bloque escalado
*/

void escaleArea(int xini,int yini,int xfin, int yfin, int xfinscaled, int yfinscaled) {
        //recorremos el bloque y vamos haciendo el nuevo
        int incx=(1+xfin-xini)/(1+xfinscaled-xini);
        int incy=(1+yfin-yini)/(1+yfinscaled-yini);


        int xi=0;
        int yi=0;
        for (int y=yini; y<=yfin;y+=incy) {
                for (int x=xini; x<=xfin;x+=incx)  {
                        //leemos el color de la imagen img_orig
                        int Y=0;
                        float numerador=0;
                        float denominador=0;
                        float centrox=(x+(incx/2)-0.5f);
                        float centroy=(y+(incy/2)-0.5f);
                        int malla=0;
                        int lado=1+xfin-xini;
                        if (lado==32) malla=2;
                        if (lado==16) malla=1;
                        int tipo=1;
                        if ((xfinscaled-xini+1==4) && (yfinscaled-yini+1==4)) tipo=0;
                        if ((xfinscaled-xini+1==8) && (yfinscaled-yini+1==4)) tipo=3;
                        if ((xfinscaled-xini+1==4) && (yfinscaled-yini+1==8)) tipo=2;

                        for (int y2=y;y2<y+incy;y2++) {
                               for (int x2=x;x2<x+incx;x2++) {
                                       double peso=tabla_pesos[malla][tipo][x2-x][y2-y];
                                       if (peso==0) {
                                            double dx=Math.abs(x2-centrox);
                                            double dy=Math.abs(y2-centroy);
                                            double d=Math.sqrt(dx*dx+dy*dy);
                                            peso=1.0f/(d);
                                            tabla_pesos[malla][tipo][x2-x][y2-y]=peso;
                                        }
                                         numerador+=lum_orig[y2*ancho+x2]*peso;
                                         denominador+=(peso);

                               }//x2
                         }//y2
                        Y=(int)(numerador/denominador);
                        if (Y>255)Y=255;
                        else if (Y<0)Y=0;

                        lum_escaled[xini+xi+(yini+yi)*ancho]=Y;
                        xi++;
                }//for x
                yi++;
                xi=0;
        }//for y
}
```

- ArrayList<Block> escaleBlock(int macroblock, int malla, String mode)

```
/**
* Escala el macrobloque
* @param macroblock macrobloque
* @param malla malla
* @param mode modo de crominancia
* @return ArrayList de Block
*/

public ArrayList<Block> escaleBlock(int macroblock, int malla, String mode) {
        // el macrobloque a escalar puede ser de cualquier malla,
        // si escale_type es 0,2,3 lo escalo a 4x4 pixels o a 4x8 o a 8x4
        // si escale_type es 1 no lo escalo y lo que hago es escalar sus bloques internos ( recursivo)

        int[] xy=getxy(macroblock,malla);
        ArrayList<Block> local_block_list = new ArrayList<Block>();


        if (escale_type[malla][macroblock]==0) {
                //se puede escalar
                escaleArea(xy[0],xy[1],xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-1,xy[0]+4-
1,xy[1]+4-1);

                //metemos este bloque en la lista de bloques ya que despues de codificar será
reescalable.
                Block  b=new  Block(xy[0],xy[1],xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-1,xy[0]+4-
1,xy[1]+4-1,0,malla, macroblock, mode);
                        block_list.add(b);
                        local_block_list.add(b);
                } else if (escale_type[malla][macroblock]==2) {
                        escaleArea(xy[0],xy[1],xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-1,xy[0]+4-
1,xy[1]+8-1);
                        Block   b=new   Block(xy[0],xy[1],xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-
1,xy[0]+4-1,xy[1]+8-1,2,malla,macroblock, mode);
                        block_list.add(b);
                        local_block_list.add(b);
                } else if (escale_type[malla][macroblock]==3)    {
                        //se puede escalar
                        escaleArea(xy[0],xy[1],xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-1,xy[0]+8-
1,xy[1]+4-1);
                        Block   b=new   Block(xy[0],xy[1],xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-
1,xy[0]+8-1,xy[1]+4-1,3,malla,macroblock, mode);
                        block_list.add(b);
                        local_block_list.add(b);
                } else {//tipo=1

                if (malla==0)  {//no escalable de malla 0
                        tamano_img_escaled+=block_len[malla]*block_len[malla];

                        for (int y=xy[1]; y<=xy[1]+block_len[malla]-1;y+=1) {
                                for (int x=xy[0]; x<=xy[0]+block_len[malla]-1;x+=1) {
                                        lum_escaled[y*ancho+x]=lum_orig[y*ancho+x];

                                }
                        }

                        Block   b=new   Block(xy[0],xy[1],xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-
1,xy[0]+block_len[malla]-1,xy[1]+block_len[malla]-1,1,malla,macroblock, mode);

                        block_list.add(b);
                        local_block_list.add(b);
```

```
                } else   {//bloque no escalable de malla>0

                        //si la malla no es cero, puede que aunque este
                        //macrobloque no sea escalable, si que lo sean sus sub-bloques
                        int b0=getB(macroblock,malla,malla-1);
                        local_block_list.addAll(escaleBlock(b0,malla-1, mode));
                        int b1=b0+1;
                        local_block_list.addAll(escaleBlock(b1,malla-1, mode));
                        int b2=b0+ancho/block_len[malla-1];

                        local_block_list.addAll(escaleBlock(b2,malla-1, mode));
                        int b3=b2+1;

                        local_block_list.addAll(escaleBlock(b3,malla-1, mode));
                }//end bloque no escalable de malla>0

        }

         return local_block_list;
}//end funcion
```

- void fillScaleType(int malla, int mb)

```
/**
 * Rellena los tipos de escalados para las diferentes mallas, excepto la malla 0 que se rellena en
compute metrics
 * TIPOS DE ESCALADO:
 * 0-> escalado
 * 1-> no se puede escalar
 * 2-> escalado 4x4
 * 3-> escalado 4x8
 * 4-> escalado 8x4
 * 5-> igual que 1 pero para vacíos
 * 6-> igual que 6 pero para vacíos
 * 7-> igual que 7 pero para vacíos
 * Un vacío es un bloque que solo se ha codificado con unos
 * @param malla malla
 * @param mb macrobloque de la malla
 */
public void fillScaleType(int malla, int mb)  {
       if (malla==0) return; //la malla 0 ya se rellena en compute metrics

       //mb es el macrobloque de la malla "Malla"
       //hay que recorrer todos sus subbloques de nivel malla-1
       //queremos calcular sus metricas a partir de las metricas de sus 4 sub-bloques
       int[] b=new int[4];

       //un macrobloque tiene 4 subbloques
       //ab
       //cd
       b[0]=getB(mb,malla,malla-1);// subbloque a
       b[1]=b[0]+1;                //subbloque b
       b[2]=b[0]+ancho/block_len[malla-1];// subbloque c
       b[3]=b[2]+1;     //subbloque d
       //inicializamos por si acaso. asi la funcion sera idempotente

       //tengo que ver si los de la malla anterior han sido calculados
       for (int i=0;i<4;i++) {
          if (Savg[malla-1][b[i]]==0) fillScaleType(malla-1,b[i]);
       }
```

```
        Savg[malla][mb]=0;
        Sh[malla][mb]=0;
        Sv[malla][mb]=0;
        //supongo que las mallas anteriores ya estan calculadas
        for (int i=0;i<4;i++) {
               Savg[malla][mb]+=    Savg[malla-1][b[i]];
               Sh[malla][mb]+=Sh[malla-1][b[i]];
               Sv[malla][mb]+=Sv[malla-1][b[i]];
         }

         //ya tenemos las metricas del macrobloque

         //OJO: LAS METRICAS de sh y sv NO ESTAN NORMALIZADAS. es simplemente la suma
         // solo estan normalizadas (valores en intervalo [0-1] en malla 0)
          int tipo=1;//inicialmente no clasificado, osea relevante
          float f=factor[malla];//1.5f;

          //casos color liso
          //no usamos los umbrales de th_sh y th_sv por que son umbrales que bajan cuando la
          //calidad sube, de modo que llega un momento en que se superan en esta condicion
          //y dan por tanto menos calidad al subir la calidad.
          //habria que usar un umbral que creciese con la calidad o bien un umbral que fuese constante
          //con la calidad. Por eso uso esta alternativa, por ser facil. un umbral constante es la malla
          //y ademas para cada malla es distinto, lo cual es lo que necesitamos
          if   ((Savg[malla][mb]<th_Savg_liso[num_hops][malla]) &&
               (Sh[malla][mb]>f*(malla)*(malla) && Sv[malla][mb]>f*(malla)*(malla))) {
               tipo=0;

           }
          //caso blanco o negro
           else if ((Savg[malla][mb]>th_Savg_pelo[num_hops][malla]) &&
               (Sh[malla][mb]<=th_Sh[num_hops][malla] && Sv[malla][mb]<=th_Sv[num_hops][malla])) {
                 tipo=0;
           }
           //caso blanco o negro 2
           else if ((Savg[malla][mb]<th_Savg_liso[num_hops][malla]) &&
               (Sh[malla][mb]<=th_Sh[num_hops][malla] && Sv[malla][mb]<=th_Sv[num_hops][malla])) {
             tipo=0;
           }
           //caso gradiente o detalles suaves
           else if ((Savg[malla][mb]<th_Savg_liso[num_hops][malla]) ) {
              if (Sv[malla][mb]<=Sh[malla][mb]) tipo=3;//H
              else tipo=2;//V
           }
          //caso pelo
          else if ((Savg[malla][mb]>th_Savg_pelo[num_hops][malla])){
             if (Sv[malla][mb]<=Sh[malla][mb]) tipo=3;//H
             else tipo=2;//V
          }
          else tipo=1;//no se escala . esta linea sobra pues se inicializa a 1
          //comprobacion para difuminados
          if ((malla==2) && (tipo==0)) {
                 int bloque_malla_0=getB(mb, num_mallas-1,0);
                 int ancho_bloques=block_len[num_mallas-1]/block_len[0];
                 int bloque_fin_0=bloque_malla_0+ancho_bloques-1+(ancho_bloques-1)*ancho/block_len[0];
                 int bloques=block_len[num_mallas-1]/block_len[0];
                 for                                                                                (int
bloque_ini=bloque_malla_0;bloque_ini<=bloque_fin_0;bloque_ini+=ancho/block_len[0]){
                         for (int b0=bloque_ini;b0<bloque_ini+bloques;b0++) {
                                 if (Savg[0][b0]>= th_Savg_liso[8][0]/20)tipo=1;
                         }
                 }
         }
          escale_type[malla][mb]=tipo;
}
```

- void interpolaBlocks(boolean bilineal, int [] encoded, int[] rescaled_final, String mode){

```
/**
* Interpola los píxeles de los bloques para reescalar la imagen
* @param bilineal true para realizar interpolación lineal, false para bicúbica
* @param encoded array de luminancias
* @param rescaled_final luminancias reescaladas finales
* @param mode modo de crominancia
*/
public void interpolaBlocks(boolean bilineal, int [] encoded, int[] rescaled_final, String mode){
    int xfin_scaled =0;
    int yfin_scaled =0;
    float offy =0;
    float offx =0;
    int yic =0;
    int xic =0;
    //vamos a rellenar la cache de bloques
    //para cada pixel, apuntamos a que bloque pertenece
    for ( int i=0;i<block_list.size();i++)  {
           Block b=block_list.get(i);
           for (int y=b.yini;y<=b.yfin_orig;y++)
           for (int x=b.xini;x<=b.xfin_orig;x++)
                  cache_blocks[x][y]=b;
    }//end for bloques


    for ( int i=0;i<block_list.size();i++) {
           Block b=block_list.get(i);
          // b.printData();
           if (b.scale_type==1 && !(mode.equals("420") || mode.equals("422"))) continue;

           xfin_scaled = b.xfin_scaled;
           yfin_scaled = b.yfin_scaled;
           offx = b.offsetx;
           offy= b.offsety;
           xic = b.xinc;
           yic = b.yinc;

           if (mode.equals("420") || mode.equals("422")) {
                  xfin_scaled = b.xfin_scaled - (1+b.xfin_scaled-b.xini)/2;
                  offx = b.offsetxcrom;
                  offy= b.offsetycrom;
                  xic = b.xincrom;
                  yic = b.yincrom;

           }

           if (mode.equals("420")) {
                  yfin_scaled = b.yfin_scaled - (1+b.yfin_scaled-b.yini)/2;
           }

           for (int ys=b.yini;ys<=yfin_scaled;ys++)
           for (int xs=b.xini;xs<=xfin_scaled;xs++) {

                  //para cada pixel scaled, hay un grupo de pixels para los que este pixel es a
                  //para cada pixel scaled, hay un grupo de pixels para los que este pixel es b
                  //para cada pixel scaled, hay un grupo de pixels para los que este pixel es c
                  //para cada pixel scaled, hay un grupo de pixels para los que este pixel es d
                  Pixel p=new Pixel();
                  p.brillo=(float) encoded[ys*ancho+xs];
                  p.xs=xs;
                  p.ys=ys;
                  p.y=(p.ys-b.yini)*yic+offy+b.yini;
                  p.x=(p.xs-b.xini)*xic+offx+b.xini;
```

```
                //ahora recorro y asigno
                  //----------- D ------------------
                  for (int y=(int)(p.y-2*(offy-0.5));y <p.y;y++){
                        for (int x=(int)(p.x-2*(offx-0.5));x<p.x;x++) {
                               if ((x>=b.xini) &&  (y>=b.yini)){
                                   pixelD[x][y]=p;

                               }
                        }
                  }
                  //----------- c ------------------
                  for (int y=(int)(p.y-2*(offy-0.5));y<p.y;y++){
                        for (int x=(int)(p.x+0.5);x<=p.x+2*(offx+0.5);x++) {
                                if ((x<=b.xfin_orig)  && (y>=b.yini)) {
                                    pixelC[x][y]=p;

                               }
                        }
                  }
                  //----------- B ------------------
                  for (int y=(int)(p.y+0.5);y <=p.y+2*(offy+0.5);y++){
                        for (int x=(int)(p.x-2*(offx-0.5));x<p.x;x++) {
                               if ( (x>=b.xini) && (y<=b.yfin_orig)) {
                                   pixelB[x][y]=p;

                               }
                        }
                  }
                  //----------- A ------------------
                  for (int y=(int)(p.y+0.5);y <=p.y+2*(offy+0.5);y++){
                        for (int x=(int)(p.x+0.5);x<=p.x+2*(offx+0.5);x++){
                               if ((x<=b.xfin_orig) && (y<=b.yfin_orig)){
                                   pixelA[x][y]=p;

                               }
                        }
                  }
            }


     }//end for bloques

    //ahora vamos a recorrer los pixels de la imagen
    //para cada pixel, tendremos que averiguar los 16 pixels de alrededor
    //con esos 16 pixels haremos la interpolacion bicubica
    //en algunos casos solo usaremos interpolacion lineal entre 4 pixels, y no
    //usaremos 16. esto ocurrira entre dos bloques escalados de modo diferente.

    //creo el objeto bicubic
    BicubicInterpolator bi=new BicubicInterpolator();

    int pos=-1;
    Pixel[] pixel=new Pixel[16];//los 16 pixeles
    double[][] pix=new double[4][4];//pixeles para el objeto bicubic

    for (int y=0;y<alto;y++) {
           for (int x=0;x<ancho;x++)  {
                  //si el pixel pertenece a un bloque no escalado, ya tenemos el brillo

                  pos++;
                  float brillo=encoded[pos];

                  Block b=cache_blocks[x][y];
```

```
        if (b.scale_type!=1 || mode.equals("420") || mode.equals("422")) {
                            pixel[0]=getPixel((float)x,(float)y,'a', encoded, mode);
                            pixel[1]=getPixel((float)x,(float)y,'b', encoded, mode);
                            pixel[2]=getPixel((float)x,(float)y,'c', encoded, mode);
                            pixel[3]=getPixel((float)x,(float)y,'d', encoded, mode);

                            //ya tenemos a b c d
                            // veamos si tiene sentido sacar los 16 pixels o con estos 4 voy a
                            //hacer interpolacion lineal. para ello comprobamos si los pixels
                            // a b c d estan o no estan alineados
                            // si los pixels estan alineados, haremos interpolacion bicubica
                            // de lo contrario, podremos hacer interpolacion lineal

                            boolean interpola_bicubic=true;

                            if ( (pixel[0].ys!=pixel[1].ys) ) interpola_bicubic=false;
                            else if ( (pixel[2].ys!=pixel[3].ys) ) interpola_bicubic=false;
                            else if ( (pixel[0].xs!=pixel[2].xs) ) interpola_bicubic=false;
                            else if ( (pixel[1].xs!=pixel[3].xs) ) interpola_bicubic=false;

                            if (bilineal==true) interpola_bicubic=false;
                            if (interpola_bicubic==false) {
                                 //vamos a interpolar linealmente estos pixels
                                 brillo=interpola(x,y,pixel);//el tiempo que consume esto no es relevante
                            } else { // interpola_bicubic es true.
                                  float shift=0.5f;//0.5f; //sirve igual con 0.5 que con 1.0, comprobado

                                  pixel[4]=pixelA[(int)(pixel[0].x-shift)][(int)(pixel[0].y-shift)];
                                  if (pixel[4]==null)
                                          pixel[4]=getPixel((float)(pixel[0].x-shift),(float)(pixel[0].y-
shift),'a', encoded, mode);
                                  if (pixel[4]==null)pixel[4]=pixel[0];

                                  pixel[5]=pixelA[(int)(pixel[0].x+shift)][(int)(pixel[0].y-shift)];
                                  if (pixel[5]==null)
                                          pixel[5]=getPixel((float)(pixel[0].x+shift),(float)(pixel[0].y-
shift),'a', encoded, mode);
                                  if (pixel[5]==null) pixel[5]=pixel[0];

                                  pixel[6]=pixelA[(int)(pixel[0].x-shift)][(int)(pixel[0].y+shift)];
                                  if (pixel[6]==null)
                                          pixel[6]=getPixel((float)(pixel[0].x-
shift),(float)(pixel[0].y+shift),'a', encoded, mode);
                                  if (pixel[6]==null)pixel[6]=pixel[0];


                                 pixel[7]=pixelA[(int)(pixel[2].x-shift)][(int)(pixel[2].y+shift)];
                                  if (pixel[7]==null)
                                          pixel[7]=getPixel((float)(pixel[2].x-
shift),(float)(pixel[2].y+shift),'a', encoded, mode);
                                  if (pixel[7]==null)pixel[7]=pixel[2];

                                  pixel[8]=pixelC[(int)(pixel[2].x-shift)][(int)(pixel[2].y+shift)];
                                  if (pixel[8]==null)
                                          pixel[8]=getPixel((float)(pixel[2].x-
shift),(float)(pixel[2].y+shift),'c', encoded, mode);
                                  if (pixel[8]==null)pixel[8]=pixel[2];


                                 pixel[9]=pixelC[(int)(pixel[2].x+shift)][(int)(pixel[2].y+shift)];
                                  if (pixel[9]==null)

pixel[9]=getPixel((float)(pixel[2].x+shift),(float)(pixel[2].y+shift),'c', encoded, mode);
                                  if (pixel[9]==null)pixel[9]=pixel[2];
```

```
                                pixel[10]=pixelA[(int)(pixel[1].x+shift)][(int)(pixel[1].y-shift)];
                                 if (pixel[10]==null)

pixel[10]=getPixel((float)(pixel[1].x+shift),(float)(pixel[1].y-shift),'a', encoded, mode);
                                 if (pixel[10]==null) pixel[10]=pixel[1];

                                 pixel[11]=pixelB[(int)(pixel[1].x+shift)][(int)(pixel[1].y-shift)];
                                 if (pixel[11]==null)

pixel[11]=getPixel((float)(pixel[1].x+shift),(float)(pixel[1].y-shift),'b', encoded, mode);
                                 if (pixel[11]==null)pixel[11]=pixel[1];

                                 pixel[12]=pixelB[(int)(pixel[1].x+shift)][(int)(pixel[1].y+shift)];
                                 if (pixel[12]==null)

pixel[12]=getPixel((float)(pixel[1].x+shift),(float)(pixel[1].y+shift),'b', encoded, mode);
                                 if (pixel[12]==null)pixel[12]=pixel[1];

                                 pixel[13]=pixelB[(int)(pixel[3].x+shift)][(int)(pixel[3].y+shift)];
                                 if (pixel[13]==null)

pixel[13]=getPixel((float)(pixel[3].x+shift),(float)(pixel[3].y+shift),'b', encoded, mode);
                                 if (pixel[13]==null)pixel[13]=pixel[3];

                                 pixel[14]=pixelD[(int)(pixel[3].x+shift)][(int)(pixel[3].y+shift)];
                                 if (pixel[14]==null)

pixel[14]=getPixel((float)(pixel[3].x+shift),(float)(pixel[3].y+shift),'d', encoded, mode);
                                 if (pixel[14]==null)pixel[14]=pixel[3];

                                 pixel[15]=pixelC[(int)(pixel[3].x+shift)][(int)(pixel[3].y+shift)];
                                 if (pixel[15]==null)

pixel[15]=getPixel((float)(pixel[3].x+shift),(float)(pixel[3].y+shift),'c', encoded, mode);
                                 if (pixel[15]==null)pixel[15]=pixel[3];

                                pix[0][0]=pixel[4].brillo;
                                pix[0][1]=pixel[5].brillo;
                                pix[0][2]=pixel[10].brillo;
                                pix[0][3]=pixel[11].brillo;
                                pix[1][0]=pixel[6].brillo;
                                pix[1][1]=pixel[0].brillo;
                                pix[1][2]=pixel[1].brillo;
                                pix[1][3]=pixel[12].brillo;
                                pix[2][0]=pixel[7].brillo;
                                pix[2][1]=pixel[2].brillo;
                                pix[2][2]=pixel[3].brillo;
                                pix[2][3]=pixel[13].brillo;
                                pix[3][0]=pixel[8].brillo;
                                pix[3][1]=pixel[9].brillo;
                                pix[3][2]=pixel[15].brillo;
                                pix[3][3]=pixel[14].brillo;

                                double dx=(double) (pixel[1].x-pixel[0].x);
                                double dy=(double)(pixel[2].y-pixel[0].y);
                                double xbi=(double)(x-pixel[0].x)/dx;
                                double ybi=(double)(y-pixel[0].y)/dy;
                                if (dx==0) xbi=0;
                                if (dy==0) ybi=0;
```

```
                                //esta es una interpolacion bicubic de tipo  catmull
                                brillo=(float)bi.getValue(pix,ybi,xbi);

                                if (brillo>255) brillo=255;
                                else if (brillo<0) brillo=0;

                            }//end else if interpola_bicubic
                        } // si type!=1 ( osea si hay escalado y por tanto hay interpolacion

                    //legamos aqui despues de interpolar o bien si scaletype=1
                    rescaled_final[pos]=(int) brillo;
                   }//for x
            }//for y
}//end funcion
```

- void interpolaBlocksVecino(int[] encoded, int[] rescaled, String modoCrominancia)

```
/**
  * Interpolado vecino cercano
  * @param encoded luminancia codificada
  * @param rescaled luminancia reescalada
  * @param modoCrominancia el modo de crominancia
  */
 public void interpolaBlocksVecino(int[] encoded, int[] rescaled, String modoCrominancia) {
         int xini=0;
         int xfin_orig=0;
         int xfin_scaled=0;
         int yini=0;
         int yfin_orig =0;
         int yfin_scaled=0;

         for ( int i=0;i<block_list.size();i++) {
                 Block b=block_list.get(i);
                 xini = b.xini;
                 yini = b.yini;
                 xfin_orig = b.xfin_orig;
                 yfin_orig = b.yfin_orig;
                 xfin_scaled = b.xfin_scaled;
                 yfin_scaled = b.yfin_scaled;

                 if (modoCrominancia.equals("420")) {
                         xfin_scaled = xini + (xfin_scaled-xini+1)/2 -1;
                         yfin_scaled = yini + (yfin_scaled-yini+1)/2 -1;

                 } else if (modoCrominancia.equals("422")) {
                         xfin_scaled = xini + (xfin_scaled-xini+1)/2 -1;

                 }

        rescaleVecino(rescaled, encoded, xini, xfin_orig, xfin_scaled, yini, yfin_orig, yfin_scaled);

         }
 }
```

- void rescaleVecino(int[] rescaled, int[] encoded, int xini, int xfin_orig, int xfin_scaled, int yini, int yfin_orig, int yfin_scaled)

```
/**
  * Permite obtener las luminancias  de un bloque reescaladas por vecino cercano
  * @param rescaled array de luminancias reescaladas
  * @param encoded array de luminancias codificadas
  * @param xini x inicial del bloque
  * @param xfin_orig x final del bloque
  * @param xfin_scaled x final del bloque escalado
  * @param yini y inicial del bloque
  * @param yfin_orig y final del bloque
  * @param yfin_scaled y final del bloque escalado
  */
 public void rescaleVecino(int[] rescaled, int[] encoded, int xini, int xfin_orig, int xfin_scaled, int
yini, int yfin_orig, int yfin_scaled) {

         int xinc=(1+xfin_orig-xini)/(1+xfin_scaled-xini);
         int yinc=(1+yfin_orig-yini)/(1+yfin_scaled-yini);


         for (int y=yfin_orig;y>=yini;y--) {
                 for (int x=xfin_orig;x>=xini;x--)  {
                         float xs=xini+((float)(x)-(float)xini+0.5f)/(float)xinc;
                         float ys=yini+((float)(y)-(float)yini+0.5f)/(float)yinc;

                         // aproximando al mas cercano
                         //-----------------------------------------------------

                         int c_malo=encoded[(int)ys*ancho+(int)xs];
                         rescaled[y*ancho+x]=c_malo;
                         continue;

                 }//for x
         }//for y
 }//end funcion
```

10.2 Encoder

10.2.1 EncoderLHE

- hazLHEMacrobloque (PerceptualRelevance pr, String mode, int[] lumin_orig, int[] crominanciaU_orig, int [] crominanciaV_orig, int ancho, int alto, int num_hops, int num_mallas, int block_len_min)

```
/**
 * Se hace LHE del macrobloque, se calculan las métricas y se realiza LHE de nuevo
 * @param pr Perceptual Relevance
 * @param mode modo de crominancia
 * @param lumin_orig array con la luminancia original
 * @param crominanciaU_orig array con la crominancia U original
 * @param crominanciaV_orig array con la crominancia V original
 * @param ancho ancho de la imagen
 * @param alto alto de la imagen
 * @param num_hops número de hops
 * @param num_mallas número de mallas
 * @param block_len_min longitud mínima del bloque
 */
```

```
public void hazLHEMacrobloque (PerceptualRelevance pr, String mode, int[] lumin_orig, int[]
crominanciaU_orig, int [] crominanciaV_orig, int ancho, int alto, int num_hops, int num_mallas, int
block_len_min) {
        int tam_macro = 8* (int)Math.pow(2, num_mallas-1); //la malla más pequeña siempre vale 8
        ANCHO = ancho;
        ALTO = alto;

        int numBloquesH = ANCHO/tam_macro;
        int numBloquesV = ALTO /tam_macro;

        int macrobloque;
        int xini;
        int yini;
        int xfin_scaled;
        int yfin_scaled;

        array_ofp_macrobloque=new int[ANCHO*ALTO];
        comp_image_macrobloque = new int [ANCHO*ALTO];
        int_img_macrobloque = new int [ANCHO*ALTO];

        comp_image=new int[ANCHO*ALTO];
        int_img=new int[ANCHO*ALTO];
        array_ofp=new int[ANCHO*ALTO];

        pr.lum_orig=lumin_orig;
        pr.lum_encoded=int_img;
        pr.hops = comp_image_macrobloque;
        kopt = new Vector<BigDecimal>();

        if (mode.equals("420") || mode.equals("422")) {
                comp_cromU = new int[ANCHO*ALTO];
                comp_cromV = new int[ANCHO*ALTO];

                int_cromU=new int[ANCHO*ALTO];
                int_cromV=new int[ANCHO*ALTO];

                cromu_ofp=new int[ANCHO*ALTO];
                cromv_ofp=new int[ANCHO*ALTO];

                pr.crominanciaU_orig = crominanciaU_orig;
                pr.crominanciaV_orig = crominanciaV_orig;
                pr.crominanciaU_encoded = int_cromU;
                pr.crominanciaV_encoded = int_cromV;
        }

        for (int j=0; j<numBloquesV; j++) {
                for (int i=0; i<numBloquesH; i++) {
                        macrobloque = j*numBloquesH+i;
                        xini = i*tam_macro;
                        yini = j*tam_macro;
                        xfin_scaled = xini + tam_macro;
                        yfin_scaled = yini + tam_macro;

                        comprime8SymbolsMacrobloque (lumin_orig, array_ofp, xini, yini, xfin_scaled,
yfin_scaled);

                        pr.computeMetrics(macrobloque);
                        ArrayList<Block> lista=pr.escaleBlock(macrobloque, num_mallas-1, mode);
                        hazLHEBloques (pr,mode,lista);
                        }
                }
        }
```

- Void hazLHEBloques (PerceptualRelevance pr, String mode, ArrayList<Block> lista)

```
/**
 * Permite realizar LHE bloque a bloque
 * @param pr Perceptual Relevancce
 * @param mode 420 ó 422, es el modo de crominancia
 * @param lista Lista de bloques en los que se ha subdividido la imagen
 */
public void hazLHEBloques (PerceptualRelevance pr, String mode, ArrayList<Block> lista){
        int [] comp_cromU_bloque;
        int [] comp_cromV_bloque;
        int[] comp_bloque;

         for(int i=0; i<lista.size(); i++) {
                int xini = lista.get(i).xini;
                int yini = lista.get(i).yini;
                int xfinscaled = lista.get(i).xfin_scaled;
                int yfinscaled = lista.get(i).yfin_scaled;
                int xfinorig=lista.get(i).xfin_orig;
                int yfinorig=lista.get(i).yfin_orig;

                //tamaño de bloque  escalado
                int anchob = xfinscaled-xini+1; // calculo el ancho y el alto del bloque
                int altob = yfinscaled-yini+1;

                //arrays de saltos del bloque
                comp_bloque = new int [anchob*altob];

                //Buscamos una k óptima
                float emin=10000000;
                error_bloque=0;
                kguay = new BigDecimal("3.0");
                BigDecimal kmin = kguay;

                for (int intentos=0;intentos<2;intentos++) {
                        error_bloque=0;
                        comprime9SymbolsBloque(pr,    comp_bloque,    comp_image,    int_img,    array_ofp,
pr.lum_escaled, pr.lum_orig, pr.rescaled_hops, xini, yini,  xfinscaled, yfinscaled, xfinorig, yfinorig);
                        if (error_bloque<emin) {emin=error_bloque; kmin=kguay;}
                        kguay = kguay.add(new BigDecimal("1.0"));
                }

                kguay=kmin;
                kopt.add(kguay);

                //Realizamos LHE del bloque llamando a la función
                comprime9SymbolsBloque(pr, comp_bloque, comp_image, int_img, array_ofp, pr.lum_escaled,
pr.lum_rescaled, pr.rescaled_hops, xini, yini,  xfinscaled, yfinscaled, xfinorig, yfinorig);

                //Guardamos hops del bloque
                lista.get(i).she_hops = comp_bloque;

                //Reescalamos para tener los píxeles de los bordes
                pr.interpolaBilinear(pr.lum_rescaled,   int_img,   xini,   xfinorig,   xfinscaled,   yini,
yfinorig, yfinscaled, true);

                //Rellena los saltos reescalados.
                pr.fillRescaledHops(lista.get(i),    lista.get(i).she_hops,    pr.rescaled_hops,    xini,
xfinorig, xfinscaled, yini, yfinorig, yfinscaled);

                //PARA LA CROMINANCIA
                if (mode.equals("422") || mode.equals("420")) {
                        int anchobCrom = anchob/2;
                        int altobCrom = altob/2;
```

```
                        if (mode.equals("422")) {
                                altobCrom =altob;
                        }
                        //y las coordenadas finales de los bloques de crominancia escalados
                        int xfinscaledCrom = xini+anchobCrom-1;
                        int yfinscaledCrom = yini+altobCrom-1;

                        //Arrays de hops del bloque
                        comp_cromU_bloque =  new int [anchobCrom*altobCrom];
                        comp_cromV_bloque = new int [anchobCrom*altobCrom];

                        //Escalamos los bloques de crominancia
                        pr.escaleAreaCrominancia ( xini,yini, xfinorig, yfinorig, xfinscaled, yfinscaled,
xfinscaledCrom, yfinscaledCrom);

                        //Realizamos LHE del bloque
                        comprime5SymbolsBloque (pr,comp_cromU_bloque, comp_cromU, int_cromU, cromu_ofp,
pr.crominanciaU_escaled, pr.crominanciaU_rescaled, pr.rescaled_hops_cromU, xini, yini,  xfinscaledCrom,
yfinscaledCrom, xfinorig, yfinorig);
                        comprime5SymbolsBloque (pr, comp_cromV_bloque, comp_cromV, int_cromV, cromv_ofp,
pr.crominanciaV_escaled, pr.crominanciaV_rescaled, pr.rescaled_hops_cromV, xini, yini,  xfinscaledCrom,
yfinscaledCrom, xfinorig, yfinorig);

                        //Guardamos los símbolos en el array de símbolos del bloque
                        lista.get(i).cromU_hops = comp_cromU_bloque;
                        lista.get(i).cromV_hops = comp_cromU_bloque;

                        //Reescalamos para tener los píxeles de los bordes
                        pr.rescaleVecinoEspecial(pr.crominanciaU_rescaled,  int_cromU,  xini,  xfinorig,
xfinscaledCrom, yini, yfinorig, yfinscaledCrom);
                        pr.rescaleVecinoEspecial(pr.crominanciaV_rescaled,  int_cromV,  xini,  xfinorig,
xfinscaledCrom, yini, yfinorig, yfinscaledCrom);


                        //Rellena saltos reescalados
                        pr.fillRescaledHops(lista.get(i),                       lista.get(i).cromU_hops,
pr.rescaled_hops_cromU,  xini,  xfinorig,  xfinscaledCrom,  yini,  yfinorig,  yfinscaledCrom);  //debería
rellenar los saltos reescalados.
                        pr.fillRescaledHops(lista.get(i),                       lista.get(i).cromV_hops,
pr.rescaled_hops_cromV,  xini,  xfinorig,  xfinscaledCrom,  yini,  yfinorig,  yfinscaledCrom);  //debería
rellenar los saltos reescalados.
                }
        }
}
```

- void comprime9SymbolsBloque(PerceptualRelevance pr, int[] comp_bloque, int[] comp_image, int[] int_image, int[] array_ofp, int[] escaled, int[] rescaled, int[][] rescaled_hops, int xini, int yini, int xfin_scaled, int yfin_scaled, int xfinorig, int yfinorig)

```
/**
 * Codifica la imagen con 9 hops
 * @param pr perceptual relevance
 * @param comp_bloque aqui se guardaran los hops del bloque
 * @param comp_image aqui se guardaran los hops de la imagen
 * @param int_image array de luminancia codificadas
 * @param array_ofp array de ofp
 * @param escaled luminancias escaladas
 * @param rescaled luminancias reescaladas
 * @param rescaled_hops hps reescalados
 * @param xini coordenada x inicial del bloque
 * @param yini coordenada y inicial del bloque
 * @param xfin_scaled coordenada x escalada del bloque
 * @param yfin_scaled coordenada y escalada del bloque
```

```
* @param xfinorig coordenada x final del bloque
 * @param yfinorig coordenada y final del bloque
*/
public   void comprime9SymbolsBloque(PerceptualRelevance pr, int[] comp_bloque, int[] comp_image, int[]
int_image, int[] array_ofp,  int[] escaled, int[] rescaled, int[][] rescaled_hops, int xini, int yini,
int xfin_scaled, int yfin_scaled, int xfinorig, int yfinorig) {
        if (cache==null) {
              cache=new double[10][2][256][1000];
              for (int yref=0;yref<=255;yref++)       {
                     for (int ki=200;ki<800;ki++) {
                           double k=((double)ki)/100.0;
                           cache[8][0][yref][ki]=Math.pow((255-yref)/k, 1/k);
                           cache[8][1][yref][ki]=Math.pow((yref)/k, 1/k);
                     }

              }
        }

        int maxofp=8;//8;
        int minofp=4;
        int ofp=maxofp;
        int ofppos=maxofp;
        int ofpneg=maxofp;

        int yref=0;
        int hop=-1;
        int hop_ant=-1;
        int emin;
        int ci=0;
        int cf=0;//color final
        int[] cf2=new int[9];
        int TAM_COMP_B =0;

        int escx=xfin_scaled-xini+1;
        int escy=yfin_scaled-yini+1;

        int esc=Math.min(escx,escy);
        int lenbx=xfinorig-xini+1; //longitud x del bloque original (sin escalados)
        int lenby=yfinorig-yini+1; //longitud y del bloque original (sin escalados). es igual que lenbx

        int stepy=lenby/escy;

        for (int y=yini;y<=yfin_scaled;y++)  {
               for (int x=xini;x<=xfin_scaled;x++)     {
                    if (x>=ANCHO) continue;
                    if(y>=ALTO) continue;

                     TAM_COMP = y*ANCHO + x;

                     if (x>0 && array_ofp [TAM_COMP-1]!=0)  ofp = array_ofp [TAM_COMP-1];
                     else  ofp = maxofp;

                     if (x==0) hop_ant = -1;
                     else if (x>xini) hop_ant = comp_image [y*ANCHO + x-1]-1;
                     else  hop_ant=rescaled_hops[x-1][y]-1;

                    //vamos a usar la luminancia escalada para codificarla
                     ci=escaled[y*ANCHO+x];


                    //comienzo de linea
                 if (x==0 && y==0) {
                     yref=ci;
```

```
 }else if (x==0 && y>0){
       //son pixels internos al bloque
       if (y>yini)     yref=int_image[(y-1)*ANCHO];
       else yref=rescaled[(y-1)*ANCHO];
 } else if ((x==ANCHO-1) && (y>0)) {
       //pixels internos
       if (y>yini) yref=1*int_image[x+(y-1)*ANCHO];
       else yref=1*rescaled[x+(y-1)*ANCHO];
 } else if ((y>yini) &&(x>xini) && x!=xfin_scaled ) {
       //estos son pixels internos. tengo que usar int_image
       yref=(4*int_image[x-1+y*ANCHO]+3*int_image[x+1+(y-1)*ANCHO])/7;
 }  else if (x==xfin_scaled && (y>0)) {
       if (y>yini) yref=(4*int_image[x-1+y*ANCHO]+3*int_image[x+(y-1)*ANCHO])/7;
       else yref=(4*int_image[x-1+y*ANCHO]+3*rescaled[x+(y-1)*ANCHO])/7;
 }  else if ((y==0)&&(x>0)) {
       if (x>xini) yref=int_image[x-1];//y=0
       else   yref=rescaled[x-1];
 } else if ( (x==xini) && x>0 && y>yini){
       int ybloque=y-yini;
       //escy contiene el escalado en y (lado ya escalado)
       if (stepy==1) {//no estoy en un bloque escalado. es un bloque normal
             yref=(4*rescaled[x-1+y*ANCHO]+3*int_image[x+1+(y-1)*ANCHO])/7;
       } else {
             int coordy_ini=yini+(y-yini)*stepy;
             int lumi=0;
             for (int coordy=coordy_ini;coordy<coordy_ini+stepy;coordy++) {

                   lumi+=rescaled[x-1+coordy*ANCHO];

             }

             lumi=lumi/stepy;
             //tengo en cuenta el pixel
             yref=(4*lumi+2*int_image[x+(y-1)*ANCHO])/6;

       }

 }else if ( (y==yini) ){//incluye x=xini
       int xbloque=x-xini;
       //escy contiene el escalado en y (lado ya escalado)
       int stepx=lenbx/escx;
       if (stepx==1) {
          if (x>xini)
                yref=(4*int_image[x-1+y*ANCHO]+3*rescaled[x+1+(y-1)*ANCHO])/7;

          else yref=(4*rescaled[x-1+y*ANCHO]+3*rescaled[x+1+(y-1)*ANCHO])/7;
       } else {
          int coordx_ini=xini+(x-xini)*stepx;
          int lumi=0;
          for (int coordx=coordx_ini;coordx<coordx_ini+stepx;coordx++) {

             lumi+=rescaled[(y-1)*ANCHO+coordx];

          }

          lumi=lumi/stepx;
          yref=lumi;
          if (x>xini) yref=(4*lumi+2*int_image[x-1+(y)*ANCHO])/6;
          else yref=(4*lumi+2*rescaled[x-1+(y)*ANCHO])/6;

       }
 }
if (yref>255) yref=255;
if (yref<0) yref=0;
```

```
        //inicio valores
        nullhop=true;
        double kini=kguay.doubleValue();

         double k=kini+(float)(ofp-4)*0.1275;//0.13

         float margen_pos=(255- yref)/128.0f;//0..2
         float margen_neg=( yref)/128.0f;//0..2

         ofppos=ofp+(int)(margen_pos*2.5f);
         ofpneg=ofp+(int)(margen_neg*2.5f);

         ofppos = ofp;
         ofpneg = ofp;

         double potencia_pos=cache[8][0][yref][(int)(k*100)];
         double potencia_neg=cache[8][1][yref][(int)(k*100)];

              // Los saltos positivos
              double s1 = ofppos*potencia_pos;
              double s2 = s1*potencia_pos;
              double s3 = s2*potencia_pos;

              //Los saltos negativos
              double s4 = ofpneg*potencia_neg;
              double s5 = s4*potencia_neg;
              double s6 = s5*potencia_neg;

              cf2[0]= yref  + (int) s3; if (cf2[0]>255) {cf2[0]=255;}
              cf2[1]= yref  - (int) s6 ; if (cf2[1]<0) { cf2[1]=0;}

              cf2[2]= yref  + (int) s2; if (cf2[2]>255) {cf2[2]=255;}
              cf2[3]= yref - (int) s5; if (cf2[3]<0) {cf2[3]=0;}

              cf2[4]= yref  + (int) s1; if (cf2[4]>255) {cf2[4]=255;}
              cf2[5]= yref  - (int) s4; if (cf2[5]<0) { cf2[5]=0;}

               //los dos pequeños
               cf2[6]=yref+ofppos;if (cf2[6]>255) cf2[6]=255;
               cf2[7]=yref-ofpneg;if (cf2[7]<=0) cf2[7]=0;

               cf2[8]=yref;//no error

               //seleccion de la mejor aproximacion
                emin=255;
                 if (Math.abs(ci-cf2[0])<=emin) {hop=0;emin=Math.abs(ci-cf2[0]);}
                 if (Math.abs(ci-cf2[1])<=emin) {hop=1;emin=Math.abs(ci-cf2[1]);}

                  if (Math.abs(ci-cf2[2])<=emin) {hop=2;emin=Math.abs(ci-cf2[2]);}
                  if (Math.abs(ci-cf2[3])<=emin) {hop=3;emin=Math.abs(ci-cf2[3]);}

                  if (Math.abs(ci-cf2[4])<=emin) {hop=4;emin=Math.abs(ci-cf2[4]);}
                 if (Math.abs(ci-cf2[5])<=emin) {hop=5;emin=Math.abs(ci-cf2[5]);}

                  if (Math.abs(ci-cf2[6])<=emin) {hop=6;emin=Math.abs(ci-cf2[6]);}
                  if (Math.abs(ci-cf2[7])<=emin) {hop=7;emin=Math.abs(ci-cf2[7]);}

                  if (Math.abs(ci-cf2[8])<=emin) {hop=8;emin=Math.abs(ci-cf2[8]);}
```

```
                error_bloque+=emin;
                //seleccion realizada

                boolean salto_pequeno=false;
                boolean salto_pequeno_ant=false;


                if ((hop==7) || (hop==6)|| (hop==8)) salto_pequeno=true;
                else salto_pequeno=false;

                if ( (hop_ant==7) || (hop_ant==6) || (hop_ant==8)) salto_pequeno_ant=true;
                else salto_pequeno_ant=false;

                if( (salto_pequeno) && (salto_pequeno_ant))  {
                        ofp=ofp-1;
                        if (ofp<=minofp) {
                                ofp=minofp;
                        }
                } else {
                        ofp=maxofp;
                }

                //asigno el color final
                cf=cf2[hop];
                if (cf<0) cf=0;
                if (cf>255 )cf=255;
                comp_image[TAM_COMP]=hop+1; //Le sumo 1 porque el original no usa el 0
                comp_bloque[TAM_COMP_B]=hop+1; //Le sumo 1 porque el original no usa 0
                int_image[TAM_COMP]=cf;
                TAM_COMP_B+=1;
                array_ofp[TAM_COMP]=ofp;
                }// for y
        }// for x
}
```

10.3 Decoder

10.3.1 DecoderLHE

- void decodeBloquesLHE (PerceptualRelevance pr, int ancho, int alto, String mode, int[] comp_image, int[] comp_cromU, int[] comp_cromV, ArrayList<Block> lista)

```
/**
 * Método para controlar el bucle de decodificación por bloques
 * @param pr PerceptualRelevance
 * @param ancho Ancho de la imagen
 * @param alto Alto de la imagen
 * @param mode Mode de crominancia
 * @param comp_image hops de la luminancia
 * @param comp_cromU hops de crominancia U
 * @param comp_cromV hops de crominancia V
 * @param lista lista de bloques para decodificar
 */
public void decodeBloquesLHE (PerceptualRelevance pr, int ancho, int alto, String mode, int[]
comp_image, int[] comp_cromU, int[] comp_cromV, ArrayList<Block> lista){
```

```
        ANCHO = ancho;
        ALTO = alto;

        int lum_inicial = pr.lum_escaled[0];
        int cromU_inicial = pr.crominanciaU_escaled[0];
        int cromV_inicial = pr.crominanciaV_escaled[0];
        int[] comp_bloque;
        int [] comp_cromU_bloque;
        int [] comp_cromV_bloque;

        array_ofp_lum = new int [ANCHO*ALTO];
        int_img=new int[ANCHO*ALTO];
        pr.lum_encoded=int_img;

        if (mode.equals("422") || mode.equals("420")) {
                array_ofp_cromU = new int [ANCHO*ALTO];
                array_ofp_cromV = new int [ANCHO*ALTO];

                int_cromU=new int[ANCHO*ALTO];
                int_cromV=new int[ANCHO*ALTO];

                pr.crominanciaU_encoded = int_cromU;
                pr.crominanciaV_encoded = int_cromV;
        }

        for(int i=0; i<lista.size(); i++) {
                int xini = lista.get(i).xini;
                int yini = lista.get(i).yini;
                int xfinscaled = lista.get(i).xfin_scaled;
                int yfinscaled = lista.get(i).yfin_scaled;
                int xfinorig=lista.get(i).xfin_orig;
                int yfinorig=lista.get(i).yfin_orig;

                //tamaño de bloque  escalado
                int anchob = xfinscaled-xini+1; //calculo el ancho y el alto del bloque escalado
                int altob = yfinscaled-yini+1;

                //Obtenemos los hops del bloque
                comp_bloque   =   getHopsBloque(comp_image,   anchob,   altob,   xini,   yini,   xfinscaled,
yfinscaled);

                //Guardamos hops del bloque
                lista.get(i).she_hops = comp_bloque;

                //Rellena saltos reescalados
                pr.fillRescaledHops(lista.get(i),    lista.get(i).she_hops,    pr.rescaled_hops,    xini,
xfinorig, xfinscaled, yini, yfinorig, yfinscaled); //debería rellenar los saltos reescalados.

                //Rellenamos el array de ofp
                fillOfp9Sym(array_ofp_lum,    comp_image,pr.rescaled_hops,    xini,    yini,    xfinscaled,
yfinscaled, xfinorig, yfinorig);

                //La k elegida en el encoder
                kguay=kopt.get(i);

                //Realizamos decode LHE del bloque llamando a la función
                decode9SymbolsBloque(pr, lum_inicial, array_ofp_lum, int_img, comp_bloque,   comp_image,
pr.lum_rescaled, pr.rescaled_hops, xini, yini,  xfinscaled, yfinscaled, xfinorig, yfinorig);

                //Reescalado especial para obtener los hops en los bordes
                pr.interpolaBilinear(pr.lum_rescaled,   int_img,   xini,   xfinorig,   xfinscaled,   yini,
yfinorig, yfinscaled, true);
```

```
                //PARA LA CROMINANCIA
                if (mode.equals("422") || mode.equals("420")) {
                //Los tamaños de los bloques de crominancia escalados dependiendo del modo de crominancia
                int anchobCrom = anchob/2;
                int altobCrom = altob/2;

                if (mode.equals("422")) {  el ancho del bloque es el mismo que el original
                altobCrom =altob;
                }

                //y las coordenadas finales de los bloques de crominancia escalados
                int xfinscaledCrom = xini+anchobCrom-1;
                int yfinscaledCrom = yini+altobCrom-1;

                //Arrays de hops del bloque
                comp_cromU_bloque  =    getHopsBloque(comp_cromU,  anchobCrom,  altobCrom,  xini,  yini,
xfinscaledCrom, yfinscaledCrom);
                comp_cromV_bloque  =  getHopsBloque(comp_cromV,  anchobCrom,  altobCrom,  xini,  yini,
xfinscaledCrom, yfinscaledCrom);

                //Guardamos los símbolos en el array de símbolos del bloque
                lista.get(i).cromU_hops = comp_cromU_bloque;
                lista.get(i).cromV_hops = comp_cromU_bloque;

                //Rellena saltos reescalados
                pr.fillRescaledHops(lista.get(i), lista.get(i).cromU_hops, pr.rescaled_hops_cromU, xini,
xfinorig, xfinscaledCrom, yini, yfinorig, yfinscaledCrom);
                pr.fillRescaledHops(lista.get(i), lista.get(i).cromV_hops, pr.rescaled_hops_cromV, xini,
xfinorig, xfinscaledCrom, yini, yfinorig, yfinscaledCrom);

                //Rellenamos el array de ofp
                fillOfp5Sym(array_ofp_cromU,comp_cromU,pr.rescaled_hops_cromU,        xini,        yini,
xfinscaledCrom, yfinscaledCrom, xfinorig, yfinorig,true);
                fillOfp5Sym(array_ofp_cromV,comp_cromV,pr.rescaled_hops_cromV,        xini,        yini,
xfinscaledCrom, yfinscaledCrom, xfinorig, yfinorig,true);

                //Realizamos decode LHE del bloque
                decode5SymbolsBloque (pr, cromU_inicial, array_ofp_cromU, int_cromU, comp_cromU_bloque,
comp_cromU,    pr.crominanciaU_rescaled,    pr.rescaled_hops_cromU,    xini,    yini,       xfinscaledCrom,
yfinscaledCrom, xfinorig, yfinorig, true);
                decode5SymbolsBloque (pr, cromV_inicial, array_ofp_cromV, int_cromV, comp_cromV_bloque,
comp_cromV,    pr.crominanciaV_rescaled,    pr.rescaled_hops_cromV,    xini,    yini,       xfinscaledCrom,
yfinscaledCrom, xfinorig, yfinorig, true);

                //Reescalamos para tener los píxeles de los bordes
                pr.rescaleVecinoEspecial(pr.crominanciaU_rescaled,     int_cromU,     xini,     xfinorig,
xfinscaledCrom, yini, yfinorig, yfinscaledCrom);
                pr.rescaleVecinoEspecial(pr.crominanciaV_rescaled,     int_cromV,     xini,     xfinorig,
xfinscaledCrom, yini, yfinorig, yfinscaledCrom);
                }
        }


        pr.borrarCache();

        if (bilineal) { //En el decoder se elige este parámetro, por defecto es bilineal = true
                pr.interpolaBlocks(false,pr.lum_encoded, pr.lum_rescaled_final, "");
                eligeBloqueVecino(pr,  pr.lum_encoded,  pr.lum_rescaled_final,  "");  //Interpolación  de
vecino si el bloque lo requiere
         } else pr.interpolaBlocksVecino(pr.lum_encoded, pr.lum_rescaled_final, "");


        if (mode.equals("422") || mode.equals("420")) {
        pr.borrarCache();
        if (bilineal) {
                pr.interpolaBlocks(false,pr.crominanciaU_encoded, pr.crominanciaU_rescaled_final, mode);
```

```
        eligeBloqueVecino(pr, pr.crominanciaU_encoded, pr.crominanciaU_rescaled_final, mode);
        } else pr.interpolaBlocksVecino(pr.crominanciaU_encoded, pr.crominanciaU_rescaled_final, mode);


        pr.borrarCache();
        if (bilineal) {
                pr.interpolaBlocks(false,pr.crominanciaV_encoded, pr.crominanciaV_rescaled_final, mode);
                eligeBloqueVecino(pr, pr.crominanciaV_encoded, pr.crominanciaV_rescaled_final, mode);
        }   else pr.interpolaBlocksVecino(pr.crominanciaV_encoded, pr.crominanciaV_rescaled_final, mode);
        }
}
```

- void decode9SymbolsBloque(PerceptualRelevance pr, int ci, int[] array_ofp,int[] int_image, int[] comp_bloque, int[] comp_image, int [] rescaled, int[][] rescaled_hops, int xini, int yini, int xfin_scaled, int yfin_scaled, int xfinorig, int yfinorig)

```
/**
* Decodifica LHE 9 símbolos
* @param pr PerceptualRelevance
* @param ci luminancia del primer píxel
* @param array_ofp array ofp a(x)
* @param int_image luminancias de la imagen
* @param comp_bloque hops del bloque
* @param comp_image hops de la imagen
* @param rescaled luminancias reescaladas (para los bordes)
* @param rescaled_hops hops reescalados
* @param xini x inicial del bloque
* @param yini y inicial del bloque
* @param xfin_scaled x final del bloque escalado
* @param yfin_scaled y final del bloque escalado
* @param xfinorig x final del bloque original
* @param yfinorig y final del bloque original
*/
        public     void  decode9SymbolsBloque(PerceptualRelevance  pr,  int  ci,  int[]  array_ofp,int[]
int_image,  int[] comp_bloque, int[] comp_image,  int [] rescaled, int[][] rescaled_hops, int xini, int
yini, int xfin_scaled, int yfin_scaled, int xfinorig, int yfinorig) {
        if (cache==null) {
                cache=new double[10][2][256][1000];
                for (int yref=0;yref<=255;yref++)       {
                        for (int ki=200;ki<800;ki++) {
                                double k=((double)ki)/100.0;
                                cache[8][0][yref][ki]=Math.pow((255-yref)/k, 1/k);
                                cache[8][1][yref][ki]=Math.pow((yref)/k, 1/k);
                        }

                }
        }
        int maxofp=8;//8;
        int ofp=maxofp;
        boolean nullhop=false;//indica si vamos a usar 9 hops o solo 8
        int ofppos=maxofp;
        int ofpneg=maxofp;

        int yref=0;
        int hop=-1;
        int hop_ant=-1;

        double [] valor_salto = new double [9];

        int escx=xfin_scaled-xini+1;
        int escy=yfin_scaled-yini+1;
        int lenbx=xfinorig-xini+1; //longitud x del bloque original (sin escalados)
        int lenby=yfinorig-yini+1; //longitud y del bloque original (sin escalados). es igual que lenbx
        int stepy=lenby/escy;
```

```
        for (int y=yini;y<=yfin_scaled;y++)  {
              for (int x=xini;x<=xfin_scaled;x++)     {
                    if (x>=ANCHO) continue;
                    if(y>=ALTO) continue;
                    TAM_COMP = y*ANCHO + x;

                    hop = comp_image[TAM_COMP]-1;

                    if (x==0) hop_ant = -1;
                    else if (x>xini) hop_ant = comp_image [y*ANCHO + x-1]-1;
                    else if (x>0) hop_ant=rescaled_hops[x-1][y]-1;

                    if (x>0  && array_ofp [TAM_COMP-1]!=0)  ofp = array_ofp [TAM_COMP-1];
                    else ofp = maxofp;


                    if (x==0 && y==0) {
                           yref=ci;
                    }else if (x==0 && y>0){
                           //son pixels internos al bloque
                           if (y>yini)     yref=int_image[(y-1)*ANCHO];
                           else yref=rescaled[(y-1)*ANCHO];
                    } else if ((x==ANCHO-1) && (y>0)) {
                           //pixels internos
                           if (y>yini) yref=1*int_image[x+(y-1)*ANCHO];
                           else yref=1*rescaled[x+(y-1)*ANCHO];
                    } else if ((y>yini) &&(x>xini) && x!=xfin_scaled ) {
                           //estos son pixels internos. tengo que usar int_image
                           yref=(4*int_image[x-1+y*ANCHO]+3*int_image[x+1+(y-1)*ANCHO])/7;
                    }  else if (x==xfin_scaled && (y>0)) {
                           if (y>yini) yref=(4*int_image[x-1+y*ANCHO]+3*int_image[x+(y-1)*ANCHO])/7;
                           else yref=(4*int_image[x-1+y*ANCHO]+3*rescaled[x+(y-1)*ANCHO])/7;
                    }  else if ((y==0)&&(x>0)) {
                           if (x>xini) yref=int_image[x-1];//y=0
                           else   yref=rescaled[x-1];
                    } else if ( (x==xini) && x>0 && y>yini){
                           int ybloque=y-yini;
                           //escy contiene el escalado en y (lado ya escalado)
                           if (stepy==1) {//no estoy en un bloque escalado. es un bloque normal
                                  yref=(4*rescaled[x-1+y*ANCHO]+3*int_image[x+1+(y-1)*ANCHO])/7;
                           } else {
                                  int coordy_ini=yini+(y-yini)*stepy;
                                  int lumi=0;
                                  for (int coordy=coordy_ini;coordy<coordy_ini+stepy;coordy++) {

                                         lumi+=rescaled[x-1+coordy*ANCHO];

                                  }

                                  lumi=lumi/stepy;
                                  //tengo en cuenta el pixel
                                  yref=(4*lumi+2*int_image[x+(y-1)*ANCHO])/6;

                           }

                    }else if ( (y==yini) ){//incluye x=xini
                           int xbloque=x-xini;
                           //escy contiene el escalado en y (lado ya escalado)
                           int stepx=lenbx/escx;
                           if (stepx==1) {
                              if (x>xini)
                                     yref=(4*int_image[x-1+y*ANCHO]+3*rescaled[x+1+(y-1)*ANCHO])/7;
                              else yref=(4*rescaled[x-1+y*ANCHO]+3*rescaled[x+1+(y-1)*ANCHO])/7;
```

```
                        } else {
                           int coordx_ini=xini+(x-xini)*stepx;
                           int lumi=0;
                           for (int coordx=coordx_ini;coordx<coordx_ini+stepx;coordx++) {
                                 lumi+=rescaled[(y-1)*ANCHO+coordx];

                           }

                           lumi=lumi/stepx;
                           yref=lumi;
                           if (x>xini) yref=(4*lumi+2*int_image[x-1+(y)*ANCHO])/6;
                           else yref=(4*lumi+2*rescaled[x-1+(y)*ANCHO])/6;
                        }
                     }

                  if (yref>255) yref=255;
                  if (yref<0) yref=0;

                  float margen_pos=(255- yref)/128.0f;//0..2
                  float margen_neg=( yref)/128.0f;//0..2

                  ofppos=ofp+(int)(margen_pos*2.5f);
                  ofpneg=ofp+(int)(margen_neg*2.5f);

                  ofppos = ofp;
                  ofpneg = ofp;
                  //inicio valores
                  double kini=kguay.doubleValue();
                  double k=kini+(float)(ofp-4)*0.1275;//0.13

                  double potencia_pos=cache[8][0][yref][(int)(k*100)];
                  double potencia_neg=cache[8][1][yref][(int)(k*100)];


                  // Los saltos positivos
                  valor_salto[0] = ofppos*potencia_pos;
                  valor_salto[1] = valor_salto[0]*potencia_pos;
                  valor_salto[2] = valor_salto[1]*potencia_pos;

                   //Los saltos negativos
                   valor_salto [3]= ofpneg*potencia_neg; //System.out.println(valor_salto[3]);
                   valor_salto [4] = valor_salto [3]*potencia_neg;
                   valor_salto [5] = valor_salto [4]*potencia_neg;

                   //No vale else if porque queremos dar prioridad al salto pequeño.
                   if (hop == 0) int_image[TAM_COMP] = yref + (int) valor_salto[2];
                   if (hop == 1) int_image[TAM_COMP] = yref - (int)valor_salto[5];
                   if (hop == 2) int_image[TAM_COMP] = yref + (int)valor_salto[1];
                   if (hop == 3) int_image[TAM_COMP] = yref - (int) valor_salto[4];
                   if (hop == 4) int_image[TAM_COMP] = yref + (int)valor_salto[0];
                   if (hop == 5) int_image[TAM_COMP] = yref - (int) valor_salto[3];
                   if (hop == 6) int_image[TAM_COMP] = yref + ofppos;
                   if (hop == 7) int_image[TAM_COMP] =yref -ofpneg;
                   if (hop == 8) int_image[TAM_COMP] =yref;

                  if (int_image[TAM_COMP]>255) int_image[TAM_COMP] =255;
                  if (int_image[TAM_COMP]<0) int_image[TAM_COMP] =0;
                  }
         }
}
```

11 ANEXO II: ARTÍCULO

Logarithmical Hopping Encoding (LHE): a low computational complexity algorithm for image compression

Jose J. García Aranda[1], Marina González Casquete[1], Mario Cao Cueto[2], Joaquín Navarro Salmerón[2], Francisco González Vidal[2]

[1] Video Architecture Department
Alcatel-Lucent
Madrid, Spain
{jose_javier.garcia_aranda, marina.gonzalez_casquete}@alcatel-lucent.com

[2] Departamento de Ingeniería de Sistemas Telemáticos (DIT)
Universidad Politécnica de Madrid (UPM)
Madrid, Spain
{mcao, navarro}@dit.upm.es

Abstract- LHE (Logarithmical Hopping Encoding) is a computationally efficient image compression algorithm that exploits the Weber-Fechner law to encode the error between colour component predictions and the actual value of such components. More concretely, for each pixel, luminance and chrominance predictions are calculated as a function of the surrounding pixels and then the error between the predictions and the actual values are logarithmically quantized. The main advantage of LHE is that although it is capable of achieving low-bit rate encoding with high quality results in terms of Peak Signal-to-Noise Ratio (PSNR) and Structural SIMilarity index (SSIM), its time complexity is O(n) and its memory complexity is O(1). Furthermore, an enhanced version of the algorithm is proposed, where the output codes provided by the logarithmical quantizer are used in a pre-processing stage to estimate the perceptual relevance of the image blocks. This allows the algorithm to downsample the blocks with low perceptual relevance, thus improving the compression rate. The performance of the proposed algorithm is especially remarkable when the bit per pixel rate is low (less than 0.2 bpp), showing much better quality, in terms of PSNR and SSIM, than JPEG and slightly lower quality than JPEG-2000 but being more computationally efficient.

Keywords- image processing; image compression; image representation

I. Introduction

There are three main concepts that set the limits for image compression techniques: image complexity [1], desired quality and computational cost. This paper presents Logarithmical Hopping Encoding (LHE) algorithm, a computationally efficient algorithm for image compression.

The proposed algorithm relies on Weber-Fechner law, which states that subjective sensation is proportional to the logarithm of the stimulus intensity [2]. LHE applies this law to prediction errors instead of the stimulus itself (in this case the original luminance and chrominance signals). More concretely, LHE estimates a luminance and chrominance prediction for each pixel (using the surrounding pixels) and then encodes the prediction error using a set of logarithmically distributed and dynamically adjusted values. As can be noted, this procedure is performed in the space domain, avoiding the need of any costly transformation to the frequency domain, and therefore reducing the computational complexity.

This approach can be enhanced by including a pre-processing stage to estimate the perceptual relevance of the image blocks. As will be explained later, the perceptual relevance estimation can be obtained, in an efficient manner, using the output codes of the logarithmical quantizer. This pre-processing stage allows the algorithm to perform Region of Interest (ROI) coding [3].

The remaining of the paper is organized as follows. Section II surveys the most relevant work related to the proposed algorithm. Section III describes the structure and the workflow of LHE. In section IV, an enhanced version of the algorithm is presented. Section V shows the main results that have been obtained in the evaluation stage of the algorithm. Finally, section VI summarizes the main contributions of this paper and outlines the future lines of work.

II. Related Work

LHE can be defined as a spatial domain image compression algorithm. In the literature of image compression, spatial domain based algorithms have been extensively studied.

In [4], a spatial domain image compression algorithm is proposed. This algorithm encodes the difference between the minimum pixel value of an m x n pixel block and the current pixel. For each block, an 11 bits header is included in order to represent the minimum value of the block (8 bits) and the number of pixels required to encode the pixel difference with respect to the minimum (3 bits).

The authors of [5] present a modified approach to the previous algorithm where the final number of bits is reduced significantly by reducing the overhead bits. In [6], a variation of the previous algorithm that encodes the difference between adjacent pixels is proposed.

In [7], a logarithmic function is used as a pre-processing stage for an image compression algorithm. This algorithm comprises four stages: logarithmic transform, neighbouring difference, repeat reduction and Huffman encoding. In this paper the logarithmic function is used to reduce the range of the difference between neighbouring pixels.

LHE also has similarities to ADPCM (Adaptive Differential Pulse-Code Modulation) [8]. ADPCM uses an adaptive predictor and an adaptive quantizer. The quantizer levels for a given pixel are generated by scaling the levels used for the previous pixel by a factor that depends on the reconstruction level used for the previous pixel. ADPCM dynamically adapts the quantizer step size to the input signal, and the set of possible unary codes are linearly distributed for each sample. However, LHE unary codes are logarithmically distributed for each sample. This change on steps distribution provides better results than ADPCM. For example, according to [8], Lena image at 1.2bpp encoded with ADPCM provides a PSNR value of 30.24dB whereas LHE provides 39.1dB.

LOCO-I [9] is the algorithm at the core of the ISO/ITU standard for lossless and near-lossless compression of continuous-tone images, JPEG-LS. LOCO-I uses the prediction of samples based on a finite subset of available past data and the context modelling of the prediction error. The purpose of this context modelling is to exploit high order structures, e.g. texture patterns, by analyzing the level of activities, such as smoothness and edginess of the neighbouring samples. This context modelling provides a probabilistic model for the prediction residual (or error signal), which can be efficiently used in combination with Golomb-Rice codes. LHE uses a similar approach, but focused on lossy image compression and taking into account the logarithmical nature of human perception.

III. LHE: Basic Algorithm

In this section, the basic algorithm of the LHE is introduced. The basic algorithm of LHE is based on the prediction of colour space values (e.g. YUV) of each pixel from the previous ones. The errors of the predicted values are encoded using a set of logarithmically distributed possible values of luminance and chrominance, which are called hops. The main blocks of the basic algorithm of LHE, grouped as LHE Quantizer, are depicted in Fig. 1. Detailed information of these blocks is provided in the following subsections. The Coder translates the hop codes into a bitstream.

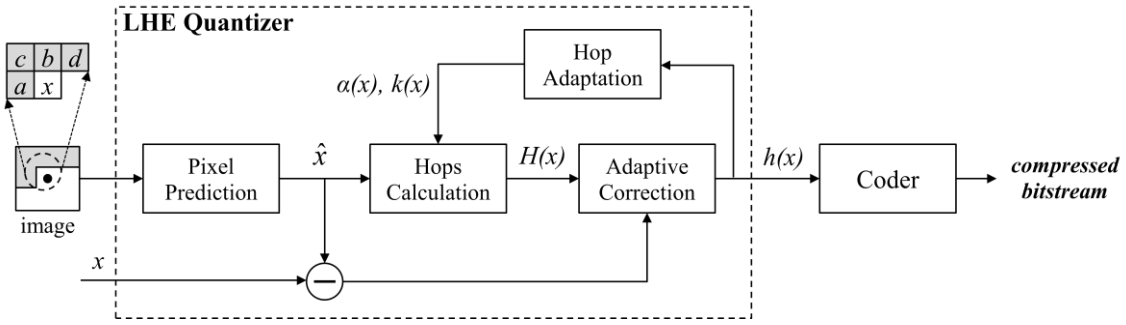


Fig 1. LHE Basic: Block Diagram

A. Pixel Prediction

LHE use the YUV colour space to represent the pixel information. YUV is defined in terms of one luminance value (Y) and two chrominance components (UV). The human eye has fairly little spatial sensitivity to colour, thus luminance component has far more impact on the image detail than the chrominance. LHE predicts the pixel values (YUV) as the average of the colour components of the top and left pixels, as in the following equation:

$$\hat{x} = \frac{a + b}{2} \quad (1)$$

The prediction of each colour component should be computed individually. Therefore, for a given pixel *x*, $\hat{x}$ represents the predicted value of the luminance (Y) or chrominance (UV). In the remaining of this section, luminance will be used as example of the three colour components.

As the predictions of the pixels depend on the previous ones, the first pixel of the image is not processed by the LHE Quantizer. Thus, its colour components are included uncompressed in order to allow the decoder to process subsequent pixels (see III.E "LHE Decoding").

B. Logarithmical Hops

As aforementioned, LHE encodes the errors of the predicted colour components from a set of possible logarithmically distributed values (called hops) for each pixel, $H(x)=\{h_{-N}, h_{-(N+1)}, ..., h_{-1}, h_0, h_1, ..., h_{N-1}, h_N\}$. The null hop h_0 means that the error associated to the predicted colour component is lower than the

one achieved by a different hop value. The smallest positive and negative hops, h_1 and h_{-1}, are not logarithmically assigned. LHE algorithm adjusts automatically, within a certain range, the value of the hops h_1 and h_{-1} for each pixel depending on the previously encoded pixel through the parameter $\alpha(x)$, which is described in the section III.D "Hop Adaptation". The first time LHE is executed, an initial fixed value for the parameter α is used, e.g. $\alpha=8$.

The following equation details the different values of the set of hops $H(x)$ for a given pixel:

$$h_i = \begin{cases} 0, & if\ i = 0 \\ \alpha(x), & if\ i = 1 \\ -\alpha(x), & if\ i = -1 \\ h_{i-1} \cdot (255 - \hat{x}/k\,(x))^{1/k(x)}, & if\ i > 1 \\ h_{i+1} \cdot (\hat{x}/k\,(x))^{1/k(x)}, & if\ i < 1 \end{cases} \quad (2)$$

The image compression rate of LHE depends on the number of hops considered ($2N+1$), the smallest non-null hops (h_1 and h_{-1}, defined by the parameter $\alpha(x)$) and the parameter $k(x)$. The higher the cardinality of H, the lower the compression rate of LHE. The parameters $\alpha(x)$ and $k(x)$ are responsible for the compactness of the set of hops $H(x)$ for a given pixel.

Different values of $k(x)$ allow expanding and shrinking the range covered by the set of logarithmical hops. In image areas where there are high component fluctuations, a low value of $k(x)$ covers the maximum range and provides better results. On the other hand, in soft detailed areas, a high value of $k(x)$ shrinks the set of hops, gaining more accuracy for small changes on colour component. The value of $k(x)$ is determined locally, at each pixel, taking into account the set of surrounding hops:

$$k(x) = f(h(a), h(b), h(c), h(d)) \quad (3)$$

For each combination of hops corresponding to the pixels a, b, c, and d, there is an optimal value for $k(x)$, which minimizes the error when a new hop for x is chosen. Although a formula could be defined, a pre-calculated table of optimal $k(x)$ values can be generated testing over all pixels from all images from an image database, and therefore setting the best values for any type of image. This strategy avoids deducing the "best logic" for the formula and therefore simplifies the problem.

C. Adaptive Correction

The Adaptive Correction module takes into account two parameters: the set of possible hops *H(x)*, computed in the previous module, and the error associated to the predicted colour component, *e*.

$$e = x - \hat{x} \tag{4}$$

The output of this module is the hop *h(x)* from the set *H(x)*, i.e. $h(x) \in H(x)$, that is closer to the above described error. In other words, the hop *h(x)* is the quantized error made by the LHE algorithm in the colour component prediction $\hat{x}$. This hop *h(x)* is the output of the LHE Quantizer.

$$h(x) = arg_{h_i} \min(|h_i - e|), h_i \in H(x) \tag{5}$$

In the particular case that there are two different hops with the same distance to the error *e*, the hop with the smaller value is chosen. The reason behind this approach is that in statistical compression of images, smaller codes are assigned to small hops (see III.E "Coder").

$$If\ \exists(h_j, h_k) \in H(x) \mid |h_j - e| = |h_k - e| \Rightarrow h(x) = h_i \mid \mathrm{i} = \min(|j|, |k|) \tag{6}$$

D. Hop Adaptation

Once the quantized error of the actual pixel is assigned, the Hop Adaptation Module updates the parameter *α(x)*, which has the same absolute colour component value as the smallest non-null hops h_1 and h_{-1}, for the next pixel. The parameter *α(x)* varies within a certain fixed range [α_{min}, α_{max}], e.g. [4, 8]. According to the equation (2) the parameter *α(x)* is used for computing the set of possible hops *H* of the next pixel. As aforementioned, an initial start value for the parameter *α* is fixed for the first pixel encoded by LHE, e.g. *α=8*.

The adjustment of the value *α* is based on the following rules:

- If the assigned hops of two consecutive pixels *{h(x-1), h(x)}* are small, i.e. they are either null hops h_0 or the smallest non-null hops $\{h_{-1}, h_1\}$, then the updated value *α(x)* becomes one unit smaller than the smallest positive non-null hop h_1, up to a certain minimum given by α_{min}.

$$If\ \{h(x-1), h(x)\} \in \{h_{-1}, h_0, h_1\} \Longrightarrow \alpha(x) = max(h_1 - 1, \alpha_{min}) \quad (7)$$

- If the quantized error *h(x)*, assigned in the previous module, is different to the null hop or the smallest non-null hops $\{h_{-1}, h_1\}$, then the updated value α is set to its maximum α_{max}.

$$If h(x) \notin \{h_{-1}, h_0, h_1\} \Longrightarrow \alpha = \alpha_{max} \quad (8)$$

D. Coder

The Coder Module translates the quantized hops of all pixels into a compressed stream of bits. One possible approach for the compression technique can be based on the existing redundancy of images across its axes, i.e. any pixel is generally similar to the previous one, and thus small hops are more frequently assigned. Although different compression techniques can be applied, this paper recommends the use of Huffman Coding algorithm. It has variable-length codes for defining the quantized errors (called hops) based on its frequency of appearance.

However, analysis over two image databases [1] [5] reveals that smallest hops are assigned in more than 90% of pixels. Therefore, in order to avoid the frequency analysis and enable real-time encoding, an effective statistical compression of hops can be achieved by assigning the smaller codes to the smaller hops. Table I shows an example of a LHE statistical coder with 5 hops codes.

Table 1. Statistical Compression for 5 hops.

Quantized Error (Hop)	Code (bits)
h_0	1
h_1	01
h_{-1}	001
h_2	0001
h_{-2}	00001

E. LHE Decoding

The LHE Decoder performs similar operations as in the LHE Quantizer but in reverse order. The following lines described the phases of the LHE Decoding process for a given pixel *x*. It should be noted that previous pixels to *x* has been already decoded and therefore the value of the parameter *α(x)* for this pixel has been already computed.

1. The binary stream is translated into symbols, in this case, into a certain hop value *h(x)*.
2. The current predicted pixel $\hat{x}$ is computed as the average of the colour components of the top and left pixels, as in equation (1).
3. Given the value of $\hat{x}$ and α, the set of hops *H(x)* for this pixel are computed by following the equation (2). At this moment, the colour component value of *h(x)* is known.
4. The decoded colour component *x'* is computed as follows:

$$x' = \hat{x} + h(x) \tag{9}$$

5. Finally, the new value for the parameter α is computed, following the rules described in section III.D Hop Adaptation.

As aforementioned, the first pixel colour components are included in raw-format in the binary stream, in order to enable the decoding process of the subsequent pixels.

Fig. 2 shows the process of encoding and decoding a figure with the Basic LHE algorithm. The LHE Quantizer assigns a symbol to each pixel (in this case, 9 hops are considered). Following the above described steps, the hop symbols are decoded as pixel colour components.

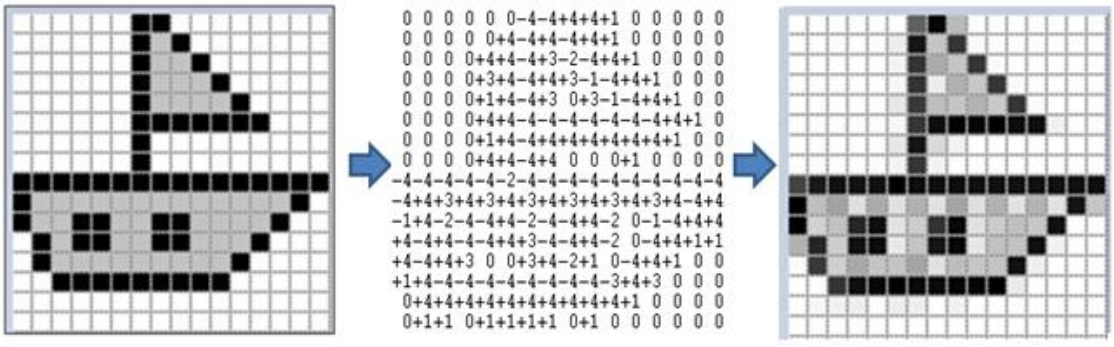


Fig 2. LHE Encoding and Decoding example.

IV. LHE: Enhanced Algorithm

A. Motivation

In [3][12], a method known as region of interest (ROI) coding is introduced. The main idea behind ROI coding is to segment an image into a region of interest and the background. If the region of interest is coded with higher fidelity than the background, a high compression ratio with good subjective quality can be achieved. ROI coding relies in the fact that the background is less perceptually relevant than the region of interest, so the coding errors made in the background are more likely to remain unnoticed than if those errors are made in the region of interest.

Analogously, LHE adopts the idea of the perceptual relevance of the different regions of an image, but instead of trying to distinguish between a region of interest and the background, LHE generalizes the ROI concept, by evaluating the perceptual relevance of each block (8x8 pixels) of the image and encoding each of these blocks accordingly.

B. Perceptual Relevance Evaluation

Intuitively, the perceptual relevance of an image block can be defined as a measure of the importance of the block regarding to the complete image, as perceived by a human viewer [13]. This subsection explains how the perceptual relevance is estimated in the LHE algorithm.

Throughout the development of the LHE algorithm, we realised that LHE-quantized luminance, i.e. the output of the LHE quantizer when using luminance as input, can be used as an estimator of the perceptual relevance of an image block. Three metrics have been defined to estimate the perceptual relevance of each block:

- S_{avg}: absolute hop index average (normalized to 0-1). This metric gives a measurement of the size of the hops. A value close to 1 indicates high luminance/chrominance fluctuations, and therefore, it suggests a complex region such as fur or sea foam, etc. It is equivalent to the entropy measurement.
- S_h: number of changes of hop index's sign when scanning the symbols horizontally (normalized to 0-1). A value close to 0 indicates that the block has low information in horizontal direction.
- S_v: number of changes of hop index's sign when scanning the symbols vertically (normalized to 0-1). A value close to 0 indicates that the block has low information in vertical direction.

In Fig. 3, these metrics are calculated over an example image. As can be seen in the example, blocks containing low information at horizontal direction, have a low S_h value. This allows detecting when a block can be down sampled horizontally. These metrics also make possible the distinction of complex regions (which have high S_{avg}, S_h and S_v values) from soft regions (with low S_{avg}, S_h and S_v values) and edges (high S_{avg} value but low S_h or S_v depending on edge direction).

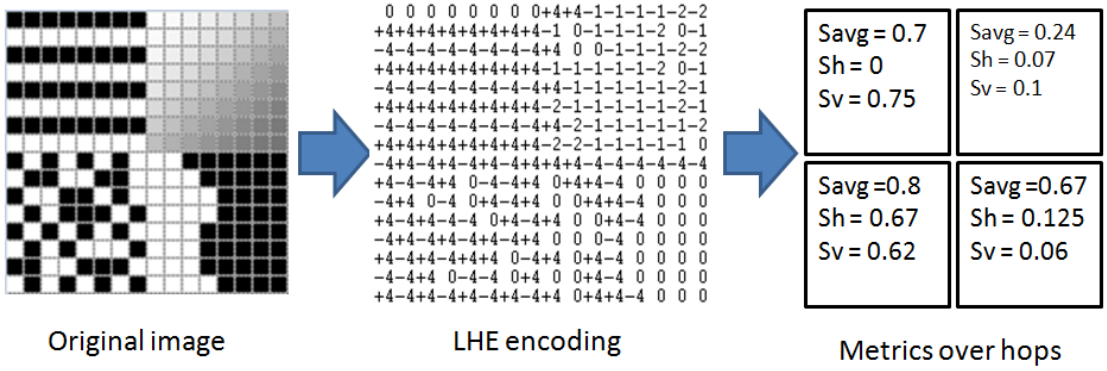


Fig 3. Perceptual relevance metrics.

The following subsections explain two methods (based on the perceptual relevance metrics) aimed at improving the performance of the basic LHE algorithm. The architecture of the enhanced algorithm is depicted in the following image.

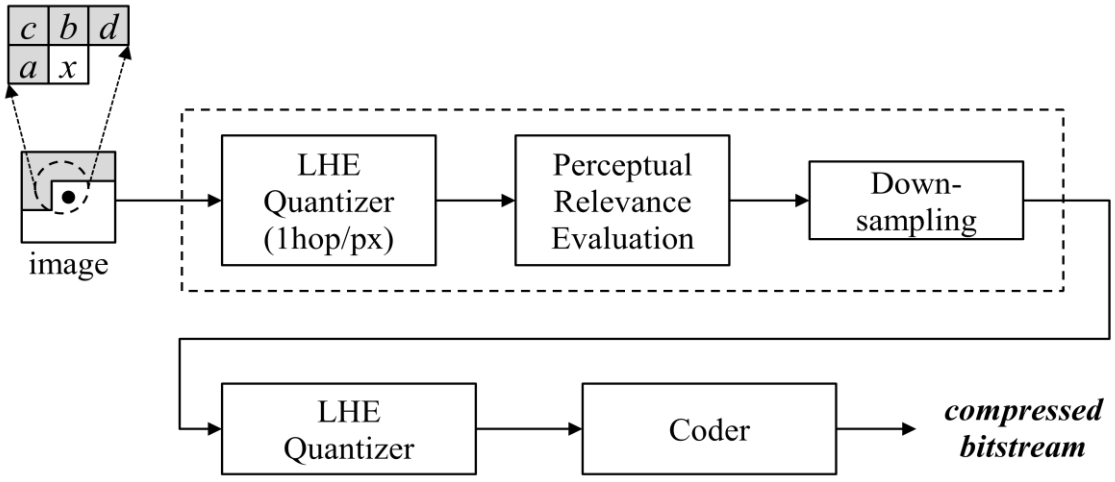


Fig 4. Enhanced LHE algorithm.

The enhanced version of the LHE algorithm uses the aforementioned perceptual relevance metrics as input for a new module: downsampling. This module is aimed to reduce the amount of information to be coded, improving the compression rate while maintaining the subjective quality. This module is described in more detail in the following subsection.

C. Downsampling

The downsampling module uses the perceptual relevance evaluation to reduce the information to be encoded. For each block, it evaluates the perceptual relevance metrics defined in the previous subsection and it decides if the block can be resized, thus reducing the number of pixels that have to be processed.

The decision to resize an image block depends on a set of thresholds that are applied to S_{avg}, S_h and S_v. More concretely, 6 thresholds are defined, a maximum and a minimum threshold for each perceptual relevance metric. Depending of the actual value of the perceptual relevance metrics with respect to the thresholds, the downsampling module can estimate the type of content of the image block, and its suitability to be resized.

The following table summarizes the detection rules that are applied to identify the type of content of an 8x8 block and the resizing strategy that is applied in each case. It also shows the binary code that is assigned to each block in order to identify how the block has been resized (vertically, horizontally or both).

Table 2. Downsampling strategies.

Type of content	Detection rule[1]	Resizing strategy	Binary code
Plain luminance	Savg↓ and Sh↓ and Sv↓	4x4 pixels (vertical and horizontal)	11
Gradated or soft details	Savg↓ and Sh↓	8x4 (horizontal)	01
	Savg↓ and Sv↓	4x8 (vertical)	10
Strong and fuzzy details, e.g. hair	Savg↑ and Sh↑	8x4 (horizontal)	01
	Savg↑ and Sv↑	4x8 (vertical)	10
Other	None Threshold is exceeded	No resizing	00

[1]↓=minimum threshold exceeded, ↑=maximum threshold exceeded

Once the image blocks have been analyzed (and resized) by the downsampling module, they will be used as input for the LHE quantizer. As can be seen, the main advantage of the downsampling process is that a certain amount of the 64 pixels blocks will be replaced by resized versions of 16 or 32 pixels.

The threshold selection establishes a trade-off between image quality and compression rate. If the thresholds are very restrictive (maximum and minimum thresholds close to 1 and 0 respectively), more quality (and less compression rate) will be achieved. More details about the threshold effect will be given in section V.

In order to take the maximum advantage of the downsampling module, a recursive procedure, using different block sizes, can be used. Let 'n' be number of iterations of this recursive procedure. First, the image is divided into macroblocks of 2^{2+n} x 2^{2+n} pixels. For each of these macroblocks, the perceptual relevance metrics are computed, and the resizing rules are checked. If the macroblock can be resized (according to the aforementioned rules) it will be resized and the next macroblock will be processed. If the macroblock cannot be resized, then it is divided into 4 blocks and the previously described downsampling technique is applied recursively to each of these inner blocks. This recursion is applied

while the block size is equal or bigger than 8x8 pixels. It should be noted that the calculation of the perceptual relevance metrics for a macroblock does not require additional processing, because the additive nature of these metrics allows them to be computed by adding the corresponding metrics of the macroblock inner 8x8 blocks. The perceptual relevance metrics for each 8x8 block are given by the perceptual relevance evaluation module, so the computational overhead in the downsampling module is low.

The following figure shows an example where the recursive downsampling procedure is applied to a 32x64 pixels image, using n=3 levels of iterations.

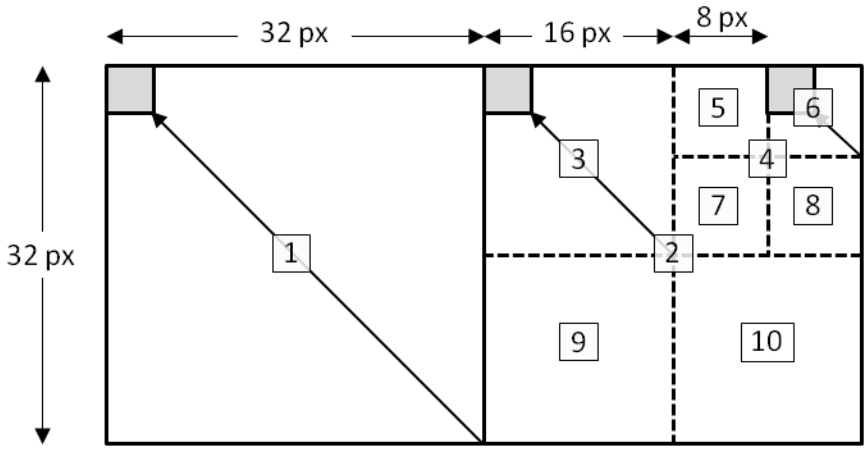


Fig 5. Recursive downsampling procedure.

The numbers enclosed in boxes represent the processing order for each block and macroblock. As can be seen, in first place, the 32x32 macroblock on the left is processed. As it can be resized, no further processing is required. Next, the 32x32 macroblock on the right is processed. This macroblock cannot be resized so its 16x16 pixels inner macroblocks will be processed. The first 16x16 macroblock (on the top and the left) can be resized, so the algorithm continues with the next one. The second 16x16 macroblock cannot be resized, so it is required to process its 8x8 inner blocks. This process continues until the complete image has been analyzed.

In the decoder side, the downsampling procedure can be easily reverted by applying an interpolation algorithm over the downsampled version of the decoded blocks in order to restore their original size.

LHE does not specify which downsampling or interpolation technique should be used. However, the selected technique will have an effect over the performance and the quality achieved by LHE. The downsampling and interpolation algorithms that have been used to evaluate the performance of LHE will be detailed in the results section.

V. Experimental Results

A. Image Quality

LHE has been tested using two image databases: Kodak Lossless True Color Image Suite [10] and USC-SIPI Image Database (miscellaneous volume) [11]. LHE provides, in most cases, a good objective quality (PSNR) but even better subjective quality, because bigger errors are located at pixels with strong contrast with surrounding ones, where subjective quality impact is minimized.

Edge information is vitally important in the perception of images [14]. However, this information is usually distorted when the image is encoded using DCT or other frequency based technique [15][16]. Fig. 6 shows a comparison between the noise generated by JPEG and LHE. In both cases, it can be seen that the LHE edges are cleaner than JPEG edges. Furthermore, LHE noise has less impact in the subjective quality compared to typical DCT-based algorithms noise due to the lack of visible artefacts at block boundaries.

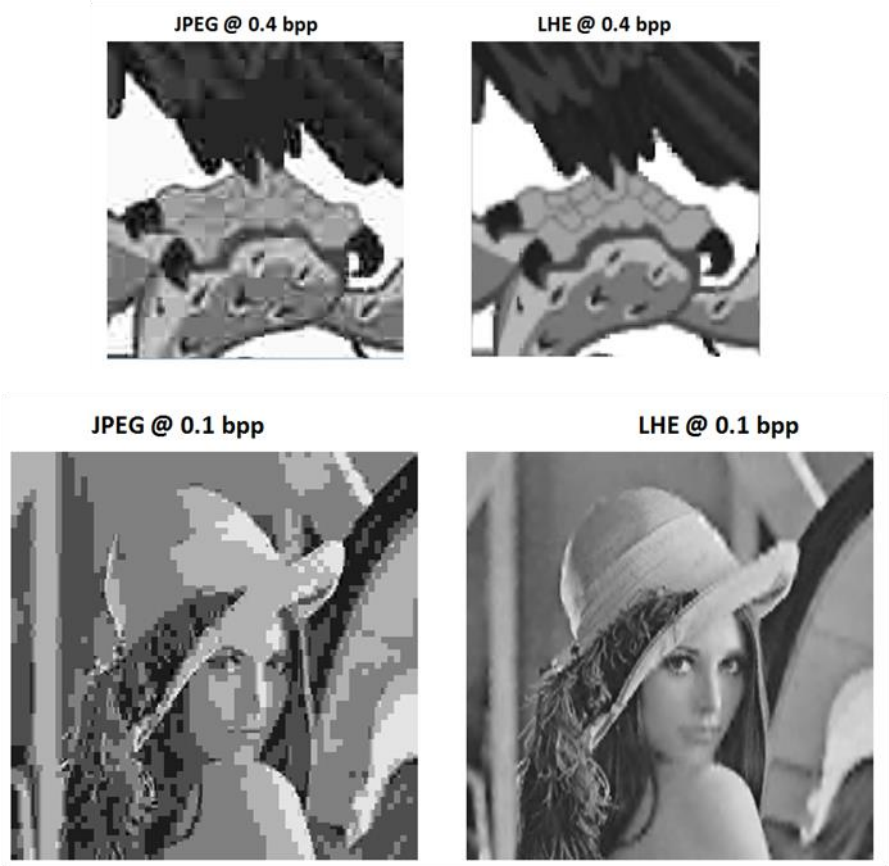


Fig 6. JPEG and LHE comparison.

The following curves are the PSNR and SSIM Rate-Distortion (R-D) diagrams for Lena image. LHE performs better than JPEG at low bitrates and provides quite similar quality at high bitrates. Regarding SSIM, LHE follows JPEG2000 trend.

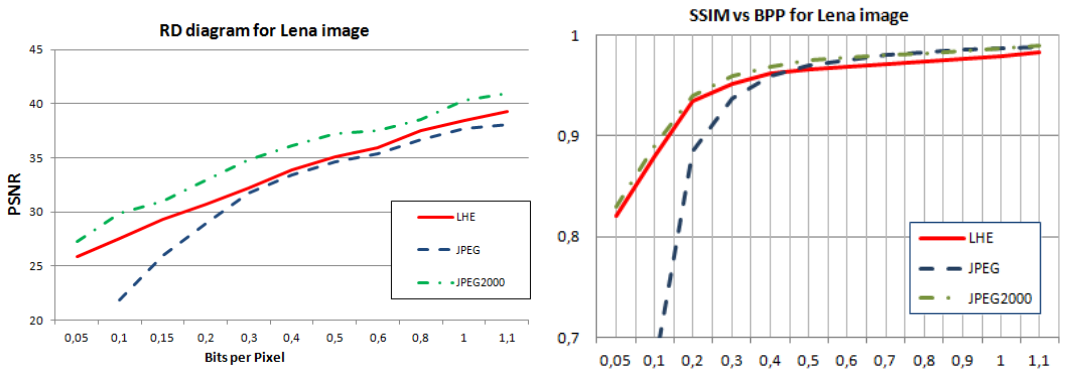


Fig 7. PSNR and SSIM R-D diagrams for "Lena" image.

LHE performance for more complex images than Lena (in terms of Spatial Information as defined in [19]) such as the image *tank 7.1.07* (USC-SIPI Image Database) is shown in the R-D diagram depicted in Fig. 8.

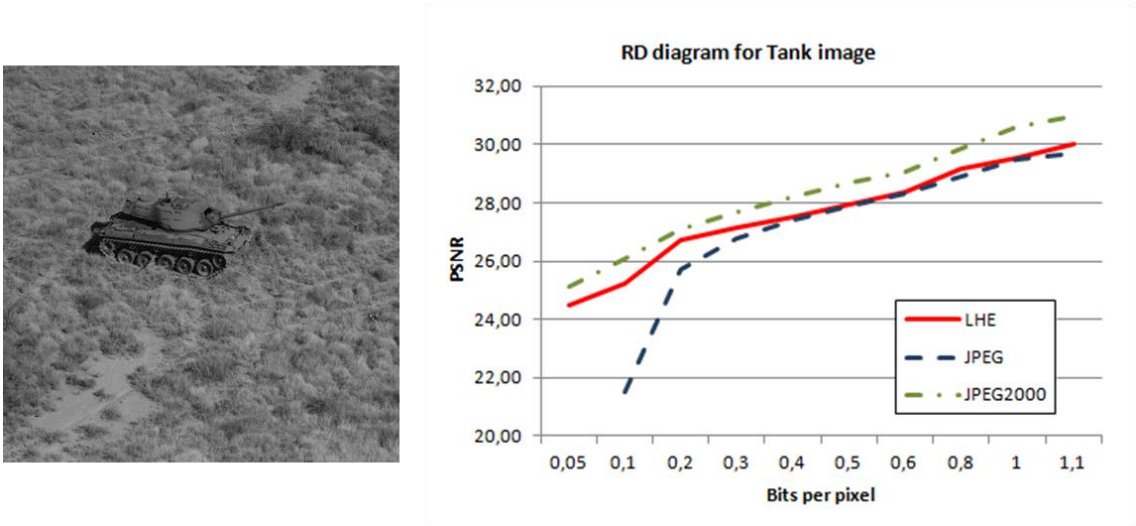


Fig 8. PSNR R-D diagram (image tank 7.1.07).

B. Algorithm Performance

One on the main advantages of LHE is its simplicity. LHE time complexity is O(n) and its memory complexity is O(1). In contrast, DCT and DWT have O(n log n) or higher computational complexity [18]. This advantage makes LHE an extremely fast procedure. In addition, LHE allows the parallel processing of those pixels which have the top and left pixels processed (needed for colour components prediction). Given an image of NxN pixels, the complete parallel process will take 2N-1 steps. In the Fig. 9 pixels are labelled with a number. Pixels labelled with the same number can be encoded in parallel (in the same step).

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

16 pixels encoded in 7 steps

Fig 9. LHE parallel processing of NxN pixels in 2N-1 steps.

This parallelization strategy, combined with the low complexity of LHE, allows very fast encoding. Our Java prototype (without parallelization) tested on Intel i5-3320@2.6Ghz encodes a 640x480 image in 15ms. Using parallelization, the encoding time can be reduced to 1ms.

This parallelization strategy can be applied to blocks instead of pixels. Thus, Enhanced LHE algorithm can take benefit from parallelization, where every block is downsampled at different ratio depending on its perceptual relevance metrics. In this case, the encoding of the blocks is made in the same order as depicted in Fig 9. The figure is referred to pixels but the same parallelization strategy can be applied to blocks composed of MxM pixels.

VI. Conclusions

LHE is a lossy compression algorithm suitable for static images based on adaptive logarithmical quantization. The main advantages of the algorithm is its low computational complexity O(n) and its performance in terms of image quality, specially at low bitrates. The algorithm design supports the parallel processing of different blocks, thus reducing the encoding time. Furthermore, LHE defines perceptual relevance metrics in order to protect the image regions of interest.

The results of this paper point to several interesting directions for future work:

- Image quality improvement based on more complex pixel prediction, downsampling strategies and interpolation techniques
- Modification of LHE algorithm for lossless image compression
- Application of LHE approach in the time-domain for video and audio compression

VII. Acknowledgement

The authors wish to thank the Spanish Science & Tech Ministry which funds this research through "INNPACTO" innovation program IPT-2011-1683-430000.

References

[1] Lloyd, S.: 'Measures of complexity: a nonexhaustive list', IEEE Control Systems Magazine, 2001, 21, (4), pp. 7-8

[2] Jayant, N., Johnston, J., Safranek, R.: 'Signal compression based on models of human perception', Proceedings of the IEEE, Oct 1993, vol.81, no.10, pp.1385-1422

[3] Cruz, D. S., Ebrahimi, T., Larsson, M., Askelof, J., Cristopoulos, C.: 'Region of Interest coding in JPEG2000 for interactive client/server applications', IEEE 3rd Workshop on Multimedia Signal Processing, 1999, pp. 389-394

[4] Hassan, S.A., Hussain, M.: 'Spatial domain lossless image data compression method', 2011 International Conference on Information and Communication Technologies (ICICT), July 2011, pp.1-4

[5] Hasan, M., Nur, K., Noor, T. B., Shakur, H. B.: 'Spatial Domain Lossless Image Compression Technique by Reducing Overhead Bits and Run Length Coding', International Journal of Computer Science and Information Technologies (IJCSIT), 2012, vol. 3, no. 2

[6] Hasan, M., Nur, K.: 'A Novel Spatial Domain Lossless Image Compression Scheme', International Journal of Computer Applications, February 2012, vol. 39, no. 15, p. 25-28

[7] Huang, S. C., Chen, L. G., Chang, H. C.: 'A novel image compression algorithm by using Log-Exp transform', ISCAS '99. Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, Jul 1999, vol.4, pp.17-20

[8] Rabbani, M., Jones, P. W.: 'Adaptive DPCM', in 'Digital Image Compression Techniques' (SPIE Press, 1991)

[9] Weinberger, M.J., Seroussi, G., Sapiro, G.: 'The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS'. IEEE Transactions on Image Processing, Aug 2000 vol.9, no.8, pp.1309-1324

[10] 'Kodak Lossless True Color Image Suite', http://r0k.us/graphics/kodak/, accessed January 2014

[11] 'USC-SIPI Image Database', http://sipi.usc.edu/database/database.php?volume=misc, accessed January 2014

[12] Christopoulos, C. A., Askelof J., Larsson, M.: 'Efficient Methods for Encoding Regions of Interest in the Upcoming JPEG 2000 Still Image Coding Standard', IEEE Signal Processing Letters, September 2000, Vol. 7, No. 9, pp. 247-249

[13] Moreno, J.M.: 'Perceptual Criteria on Image Compression', Ph.D. dissertation, DCC, UAB, Barcelona, Spain, 2011

[14] Zhang, C. N., Wu, X.: 'A hybrid approach of wavelet packet and directional decomposition for image compression', IEEE Canadian Conference on Electrical and Computer Engineering, Edmonton, Alta., Canada, Dec. 1999, vol. 2, pp. 755-760

[15] Zhou, Z., Venetsanopoulos, A. N.: 'Morphological methods in image coding', IEEE International Conference on Acoustics, Speech, and Signal Process. (ICASSP-92), San Francisco, CA, USA, March 1992, vol. 3, pp. 481-484

[16] Popovici, I., Withers, W.D.: 'Locating Edges and Removing Ringing Artifacts in JPEG Images by Frequency-Domain Analysis', IEEE Transactions on Image Processing, May 2007, vol.16, no.5, pp.1470-1474

[17] ITU-T P.910: 'Subjective Video Quality Assessment Methods for Multimedia Applications', 2008

[18] Kok, C. W.: 'Fast algorithm for computing discrete cosine transform', IEEE Transactions on Signal Processing, 1997, 45, (3), pp 757-760